Mastering TensorFlow 2.x

Implement Powerful Neural Nets across Structured, Unstructured datasets and Time Series Data

RAJDEEP DUA

Mastering TensorFlow 2.x

Implement Powerful Neural Nets across Structured, Unstructured datasets and Time Series Data

RAJDEEP DUA

Mastering TensorFlow 2.x

Learn How to Use TensorFlow Library to Solve Problems Involving Structured and Unstructured Data

Rajdeep Dua

www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91392-222

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

www.bpbonline.com

Dedicated to

My beloved Parents

Wife ***Manju*** *and Sons* ***Navtej*** *and* ***Kairav***

About the Author

Rajdeep Dua is a Senior Director at one of the leading SAAS software vendors leading teams working on Machine Learning and Deep Learning use cases. He has been tracking the TensorFlow library for the last 3-4 years. He has written multiple books in the areas focussing on TensorFlow, Neural Networks, and Apache Spark. Rajdeep's mainstream job is focused on getting AI to solve industry-specific problems and is quite familiar with the challenges faced. He has been in the industry for over 20 years working in areas like Cloud computing, Big Data, and Machine Learning.

About the Reviewer

Anitha Vijayakumar is the Director for ML Infrastructure on the TensorFlow team. She got her engineering degree from Bengaluru, India, and a Master's in Computer Engineering from UCLA. She has spent the last decade at Google working on infrastructure projects across data center deployments and machine learning. She has worked in multiple companies like Ericsson, Alcatel building software infrastructure in these organizations.

Acknowledgement

First and foremost I would like to thank my mother, wife for their continuous support and encouragement. Also, I would like to thank my two sons who are quite accommodating in the times I am not available.

I am also grateful to the wider TensorFlow community and the tech writers of the TensorFlow site which was a great source of getting started on a particular topic. I also took a lot of inputs from courses on Coursera, and some Reinforcement Learning content online learning (@Coursera by deeplearning.ai; @Udacity by Google)

My gratitude also goes to the team at BPB Publications for being supportive and providing me quite a long time to finish the book where the underlying stack version is changing every three months. When I started writing the book we were at TensorFlow 2.1 and by the time I finished, we are at 2.7.

Preface

This book covers many parts of deep learning concepts with an implementation based on TensorFlow. We will start with the basics and then delve into advanced topics and model tuning as well. The key focus of the book is to introduce a concept, the math behind it, and how the concepts can be used in the TensorFlow library. Most of the samples are based on Jupyter with some of them also based on the Google Colab environment. Book is a healthy combination of theory as well as samples to get you started. Some chapters have taken real-world examples from Kaggle – for example using the PCAM dataset for image classification as well as transfer learning. In some cases, we have also shown how to generate a synthetic dataset for a problem, for example for a time series dataset. While sufficient effort has been made to explain the topics it is almost impossible to do deep dive on each topic. Considering the trade-off between the vast majority of topics to be covered in the book and keeping the volume manageable, we have focused on applications in some cases.

The primary target of the books is to give an overview to developers and data scientists starting on TensorFlow. It also covers advanced topics on TensorFlow like Distributed TensorFlow and optimization techniques.

The reader should have some knowledge of Python as a programming language and better have some programming experience. Also, it will be good for the reader to have some basic knowledge about machine learning and some basic math background to get the most out of the book.

This is the first part of the book, and it will consist of the following four chapters, in which you will learn the following:

Chapter 1 will cover the introduction to TensorFlow. It will show how to install TensorFlow, then look at some of the building blocks of TensorFlow high-level APIs which are based on Keras. Then the chapter will help users write their first sample on TensorFlow. The chapter will also cover low-level APIs as well as image classification examples.

Chapter 2 will show how to classify structured and unstructured text as well as non-textual data using Tensorflow 2.x Keras APIs. You will also get to learn to handle the overfitting and underfitting of models.

Chapter 3 introduces Keras APIs in-depth and their application with examples. We will compare how these APIs are exposed -- for example optimizers, loss functions, and the underlying deep learning theory.

Chapter 4 will cover Convolutional networks and how we can build these kinds of neural networks to classification images with Tensorflow 2.x. It will also cover what is TFHub and how it can be used for text and image processing.

Chapter 5 will introduce the concepts of Recurrent Neural Network (RNN), Gated Recurrent Unit (GRU), Long-Term Short-Term Memory (LSTM), and bi-directional LSTM.

Chapter 6 will focus on time series forecasting using Recurrent Neural Networks (RNN) and Long Term Short Term Memory (LSTM).

Chapter 7 will provide details on how to distribute training over multiple CPUs, GPUs, and machines, it also covers building data pipelines using tf.data. We will look at how to create a Dataset from tensors. It will provide details on iteration mechanisms available to access data in tf.data.Dataset.

Chapter 8 focuses on reinforcement learning and its implementation which is specific to Tensorflow 2.0 Keras APIs. You will implement a sample where the network learns how to play Cartpole and Atari games. You will also look at various architectures like DQN and DDQN.

Chapter 9 will provide details on how to optimize the models built using various techniques like weight pruning, quantization. It will also show how to convert the Tensorflow 2.0 model to TFLite.

Chapter 10 will provide details on how to implement Generative Adversarial Networks (Simple GAN, DCGAN, and WGAN using TensorFlow 2.x.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/82a809

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Mastering-TensorFlow-2.x**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications’ Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book

Table of Contents

CHAPTER 1
Getting Started with TensorFlow 2.x

Introduction

TensorFlow has released version 2.7 (in the *2.x* series) of the API. In the first chapter, we will understand the thought process and get started with this API.

Keras is a popular deep learning API for building and training models. It is mostly used for proto-typing as well as research. The TensorFlow team has officially adopted Keras as a first-class citizen with some small modifications. The goal is to reduce confusion for developers who want to use Keras. The TensorFlow team wanted to focus on expanding the advanced capabilities for researchers.

Structure

In this chapter, we will cover the following topics:

- Installing TensorFlow 2.x
- Keras and high-level APIs integration into TensorFlow
- Writing the first sample using TensorFlow 2.x
- Low-level APIs
- Image classification with CNN

Objective

In this chapter, we will learn how to get started with TensorFlow 2.x (*x* means version *6* at the time of writing this book).

Installing TensorFlow 2.x

TensorFlow 2.x can be installed in multiple ways; but the easiest way is to install using **pip**. It can be run on Ubuntu/MacOS or Windows.

Installation on Ubuntu

Assuming you have CIT pip>= 0.19 and Python 2.7 installed on your devices, follow the given procedures. We will assume Ubuntu OS.

```
pip install —upgrade tensorflow
```

Notice how TensorFlow 2.6 is installed by default.

Testing the TensorFlow installation: Run the following command to check whether it is installed as expected:

```
python -c "import tensorflow; print(tensorflow.__version__)"
```

The response should be:

```
2.7.0
```

For Python 3.4 or above, the command is similar to the following:

```
pip install —upgrade tensorflow
```

Installing TensorFlow 2.x with GPU support

In case you want to install TensorFlow with GPU support, use the following command :

```
pip install tensorflow-gpu
```

Given that our environment setup is complete, let us look at high-level Keras APIs before delving deeper into TensorFlow components.

Keras high-level APIs integration into TensorFlow

Keras high-level APIs has the following advantages over traditional TensorFlow 1.x based flow:

- **User-friendly**: API is developer-friendly and easy to understand. This allows easier composition of neural networks.
- **Easily composable**: This is enabled with the concept of sequential models for beginners as well as for experts. APIs are not only simple enough to be used by the beginners but are also flexible with low-level functional APIs to create more complex flows.

- **Difference between Keras and TensorFlow 2.x**: Keras is a high-level API from a built-in sequential model perspective. It has the ability to define customs. *Figure 1.1* compares TensorFlow and Keras on two dimensions: architectural and training APIs:

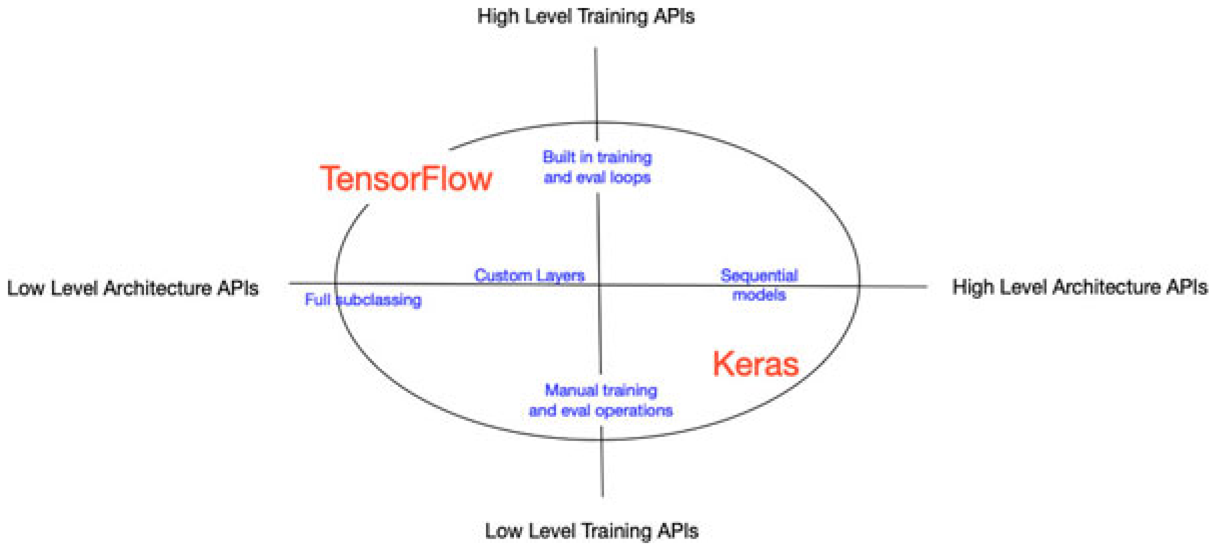

High Level Training APIs
TensorFlow
Built in training and eval loops
Low Level Architecture APIs
Custom Layers
Sequential models
High Level Architecture APIs
Full subclassing
Keras
Manual training and eval operations
Low Level Training APIs

Figure 1.1: *Comparing TensorFlow and Keras from an architecture and training APIs perspective*

Keras' network topology uses custom layers, but it is not as sophisticated as TensorFlow's graph creation low-level APIs.

Python binding

Python is the most complete language binding in TensorFlow, and we will use it as our primary language.

All the tensors are statically typed since the underlying implementation in TensorFlow is in C++. Python bindings are able to inspect and show the underlying data type.

Functional Spec

Keras has three main objects:

- **Keras tensor**: This is an augmented version of TensorFlow tensor.
- **Layer:** This helps to perform transformation on tensors.
- **Model**: This is a specification of a neural network and associated loss functions, optimizers, and so on.

Keras tensor

Keras tensor is generated by the **`Input`** function as follows:

```
from tensorflow.keras.layers import Input
x = Input(batch_shape=(100, 100))
```

A Keras tensor is a tensor object from the underlying backend (TensorFlow), which is augmented with attributes that allow us to build a Keras model just by knowing the inputs and outputs of the model:

```
x
   <tf.Tensor 'input_1:0' shape=(100, 100) dtype=float32>
```

```
type(x)
tensorflow.python.framework.ops.Tensor
```

```
vars(x)
{'_op': <tf.Operation 'input_1' type=Placeholder>,
 '_value_index': 0,
 '_dtype': tf.float32,
 '_tf_output':
 <tensorflow.python.pywrap_tensorflow_internal.TF_Output;
 proxy of <Swig Object of type 'TF_Output *' at
 0x7feab8370690> >,
 '_shape_val': TensorShape([100, 100]),
 '_consumers': [],
 '_id': 2,
 '_name': 'input_1:0',
 '_keras_history': KerasHistory(layer=
 <tensorflow.python.keras.engine.input_layer.InputLayer object
 at 0x7feb09827fd0>, node_index=0, tensor_index=0),
 '_keras_mask': None}
```

A Keras tensor **`x`** has the same type as a TensorFlow tensor, as we can see below. However, what makes **`x`** a Keras tensor is the existence of Keras-specific attributes, such as, **`_keras_history`**.

Layer

A layer defines a transformation in the network. The layer accepts **`tf.keras`** tensor(s) as input, transforms the input(s), and outputs Keras tensor(s). Layers can do a wide variety of transformations. Dense, activation, reshape, Conv2D, and LSTM are all layers derived from the abstract **`layer`** class. The following figure shows the relationship between **Tensor** and a

Layer:

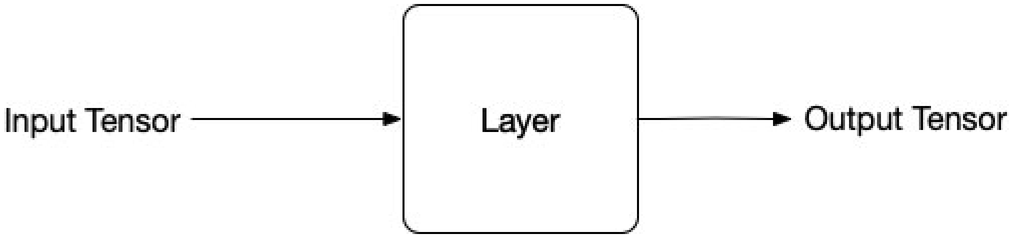


Figure 1.2: *Layer's input and output parameters*

Let us see how to create a simple dense layer in TensorFlow 2.x:

```
from tensorflow.keras.layers import Dense
dense_layer = Dense(units=10, activation='relu')
dense_layer
<tensorflow.python.keras.layers.core.Dense at 0x7feaa9f19150>
```

Let us look at the underlying type:

```
tensorflow.python.keras.layers.core.Dense
```

In the preceding code snippet, **`dense_layer`** is an object of the class **`Dense`**. The **`Layer`** objects are callable because of the **`__call__`** method. The **`__call__`** method accepts a tensor or a list/tuple of tensors and returns them. The **`__call__`** method can only accept tensors of shapes which are compatible with its object.

A layer may or may not have weights associated with it; it depends on what it does. A Dense layer (a subclass of layer) does have weights associated with it. When using the functional API, the weights are not instantiated until the **`dense_layer.__call__()`** method is called.

Keras graph

While using the backend, a graph is built to describe the computations intended to be performed. This graph can be implicitly created when doing eager execution, or explicitly (in TensorFlow 1.x using **`session.run()`**).

Though the graph creation is hidden in Keras, it still relies on it for computations. Graph creation can be fine-tuned by using Keras functional APIs.

Writing the first sample using TensorFlow 2.x

In this section, we look at how to write a simple model to enable image classification based on **cifar-10** datasets.

This is a fast-paced overview of the complete TensorFlow program with the details explained as you go. We will use the **`tf.keras`** to build and train models in TensorFlow.

Here, *50,000 images* are used to train the network and *10,000 images* to evaluate how accurately the network learned to classify images. You can access the **cifar-10** directly from TensorFlow. **`import`** and load the **cifar-10** dataset directly from TensorFlow:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Loading the dataset returns for NumPy arrays: the **`train_images`** and **`train_labels`** arrays are the training set—the data the model uses to learn. The model is tested against the test set, the **`test_images`**, and **`test_labels`** arrays.

The images are *32x32x3* NumPy arrays, with pixel values ranging from *0* to *255*. The labels are an array of integers, ranging from *0* to *9*. These correspond to the class the image represents.

Preprocess the data

The data must be preprocessed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of *0* to *255* as shown in *Figure 1.3*:

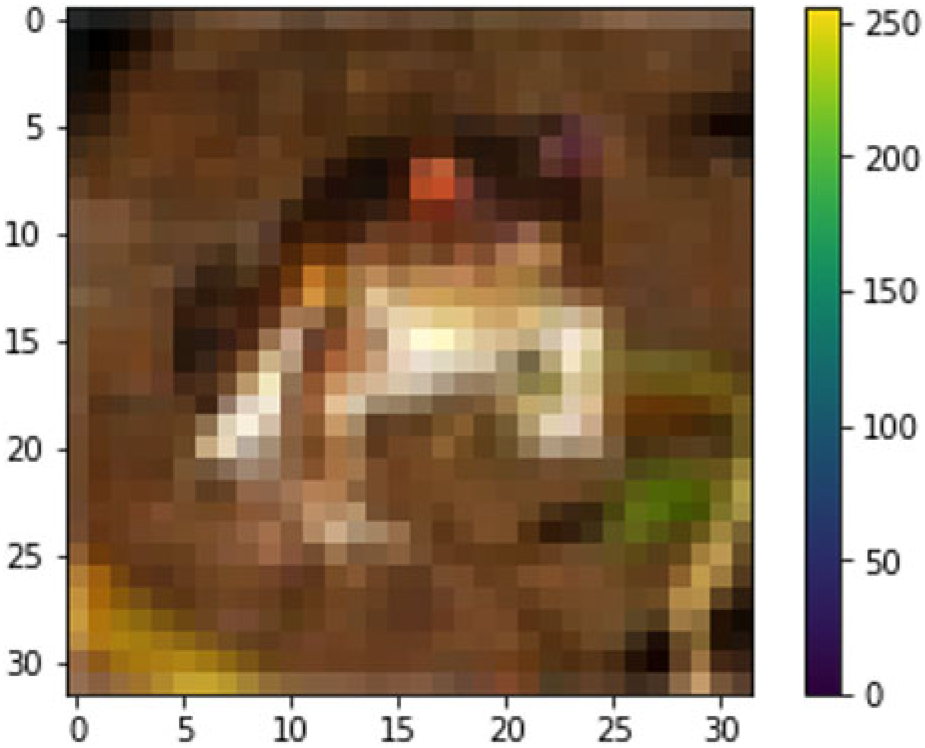


Figure 1.3: *Image with actual pixel values*

First, we will normalize the images by dividing them by `255`:

```
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images
/ 255.0
```

Figure 1.4 shows how the pixel values get transformed:

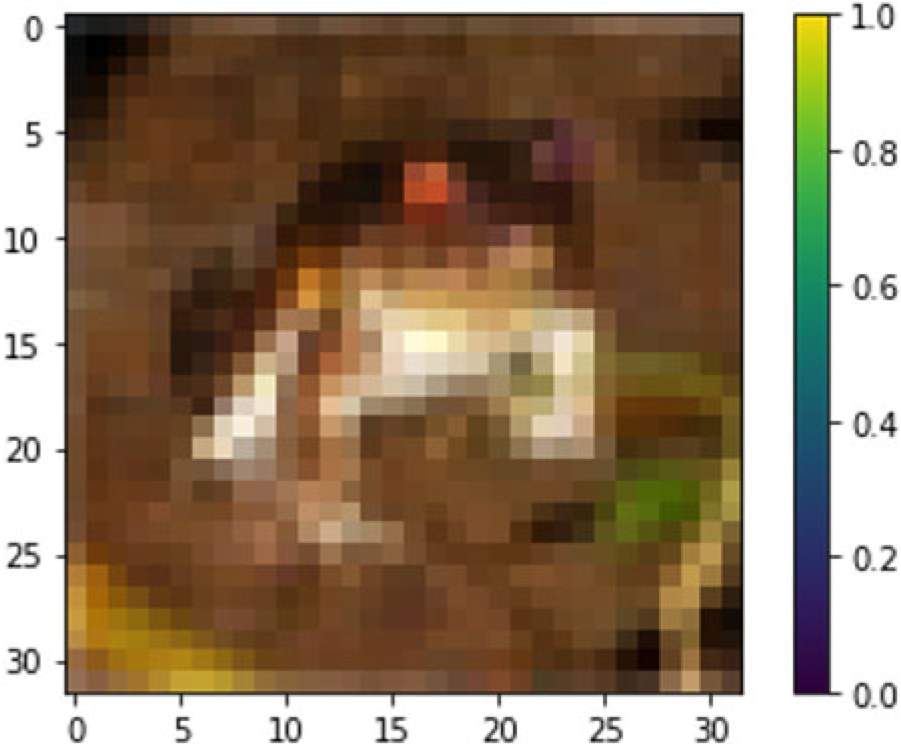


Figure 1.4: *Normalized pixel values*

Each image is mapped to a single label. Since the class names are not included with the dataset, store them here to use later when plotting the images:

```
class_names = ['airplane', 'automobile', 'bird', 'cat',
'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

Let us plot the images with the labels in a *3x3* grid:

```
plt.figure(figsize=(5,5))
for i in range(9):
  plt.subplot(3,3,i+1)
  plt.xticks([])
  plt.yticks([])
  plt.grid(False)
  plt.imshow(train_images[i], cmap=plt.cm.binary)
  # The CIFAR labels happen to be arrays,
  # which is why you need the extra index
  plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

The output of the preceding plot is a *3x3* grid with labels as shown in *Figure 1.5*:

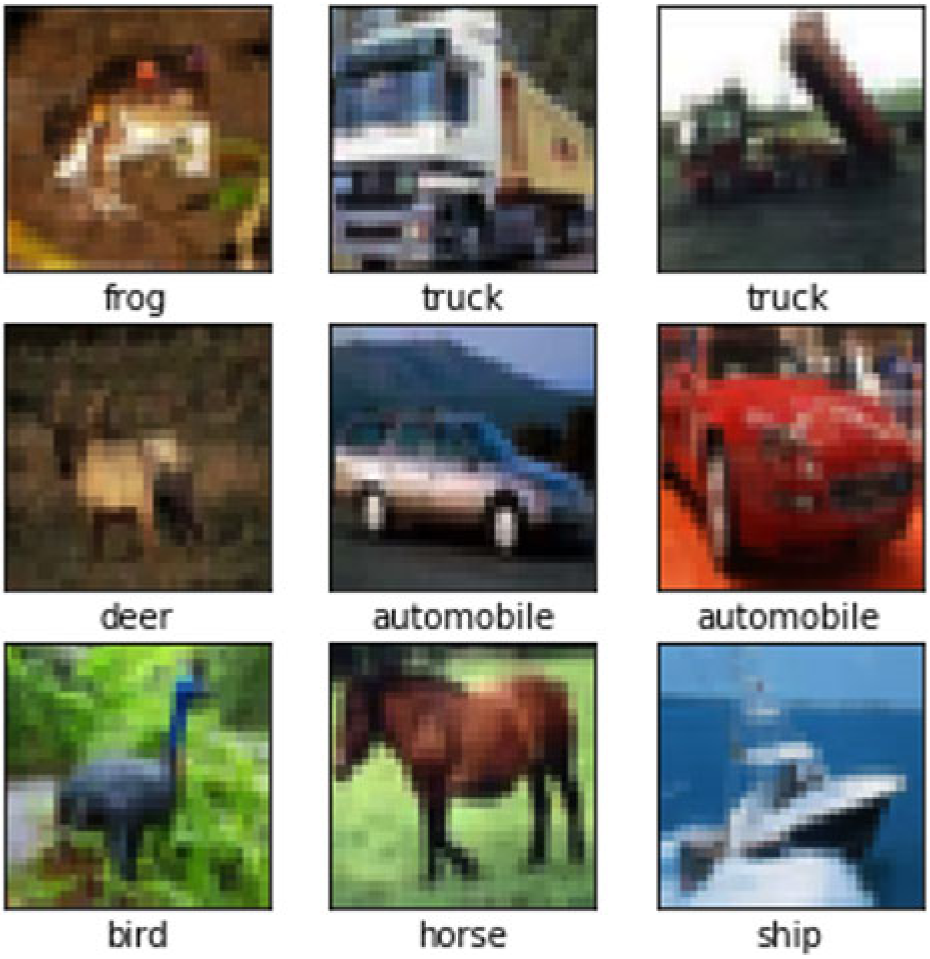


Figure 1.5: *3x3 matrix of training images with labels*

We have normalized the data and seen how it looks. Now, we need to work on creating the model using layers.

Build the model

To build the model, we will first configure the layers of the model and then compile it.

Setting up the layers

The layer is the basic building block and neural network. Layers extract representations from data that is fed to them. Most of the deep learning model's creation consists of chaining together simple layers. Layers, like

`tf.keras.layers.Dense`, have hyper-parameters that are learned during training.

We created a basic sequential model and added three layers to it:

- The first layer flattens the input from **`(?, 32x32x3)`** to **`(?, 3072)`**, a two dimensional tensor. After the data is the flattened, the network has two dense layers. These are fully connected neural layers.
- The first layer has **`64`** nodes and the second layer has **`10`** node **`softmax`** layers, which returns an array of **`10`** probability scores that add up to **`1`**.
- Each node indicates a probability that the current images belong to one of the **`10`** classes:

  ```
  model = models.Sequential()
  model.add(layers.Flatten(input_shape=(32, 32, 3)))
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(10, activation='softmax'))
  model.summary()
  ```
- Once the model is created, we need to compile and train the model.

Compile the model

Before the model is ready, it needs to be compiled keeping in mind the following information:

- **Loss function**: It calculates how accurate the model is during training, for example, **`sparse_categorical_crossentropy`**.

  The formula for categorical cross entropy is as follows:

  $$-\frac{1}{N}\sum_{s\in S}\sum_{c\in C}1_{s\in c}\,logp(s\in c)$$

  - ***S***: Samples.
  - ***C***: Classes.
  - ***s*∈*c***: Sample belongs to class ***c***.

    For cases when classes are exclusive, no need to sum over them. For each sample, only non-zero value is $-log(s\in c) - logp(s\in c)$ for true class c.

- **Optimizer**: It defines how models are updated based on the data it sees and its loss functions. Examples are **Adam**, **AdaDelta**, **Adagrad**, and so on. In the upcoming chapters, we will learn more about these optimization techniques.
- **Metrics**: It is used to monitor training and validation/testing steps. In our example, we will use **accuracy**. Accuracy is the fraction of images correctly classified:

  ```
  model.compile(optimizer='adam',
       loss='sparse_categorical_crossentropy',
       metrics=['accuracy'])
  ```

After having compiled the model, it is ready for training using the training data.

Train the model

Training the model is done using the following steps:

- The model is fed the training data, which is **`train_images`** and **`train_labels`** arrays.
- The model learns from images and labels.
- The model is used to make predictions on test set.
- Verify that the predictions made by the model match the labels.

Feed the model

1. To start the training, a call is made to the **`model.fit`** method—it *fits* the model to the training data:

   ```
   EPOCHS = 10
   history = model.fit(train_images, train_labels,
   epochs=EPOCHS, validation_data=(test_images, test_labels))
   ```

   The output will show a combination of accuracies and losses for **`10`** epochs:

   ```
   Train on 50000 samples, validate on 10000 samples
   Epoch 1/10
   50000/50000 [==============================] - 4s
   77us/sample - loss: 1.9230 - accuracy: 0.3086 - val_loss:
   1.8549 - val_accuracy: 0.3165
   ```

```
Epoch 2/10
50000/50000 [==============================] - 3s
65us/sample - loss: 1.8041 - accuracy: 0.3568 - val_loss:
1.7702 - val_accuracy: 0.3636
Epoch 3/10
50000/50000 [==============================] - 3s
64us/sample - loss: 1.7532 - accuracy: 0.3771 - val_loss:
1.7386 - val_accuracy: 0.3860
Epoch 4/10
-----
Epoch 8/10
50000/50000 [==============================] - 3s
65us/sample - loss: 1.6767 - accuracy: 0.4038 - val_loss:
1.6715 - val_accuracy: 0.4037
Epoch 9/10
50000/50000 [==============================] - 3s
64us/sample - loss: 1.6679 - accuracy: 0.4050 - val_loss:
1.6598 - val_accuracy: 0.4041
Epoch 10/10
50000/50000 [==============================] - 3s
67us/sample - loss: 1.6585 - accuracy: 0.4097 - val_loss:
1.6474 - val_accuracy: 0.4080
```

2. Let us plot these values in a plot for accuracy and validation loss:

```
import sys; sys.path.append('..')
from common.plot_util import eval_metric
import matplotlib.pyplot as plt
def eval_metric(model, history, metric_name, EPOCHS):
 '''
 Function to evaluate a trained model on a chosen metric.
 Training and validation metric are plotted in a
 line chart for each epoch.
 Parameters:
  history : model training history
  metric_name : loss or accuracy
 Output:
  line chart with epochs of x-axis and metric on
  y-axis
```

```
'''
metric = history.history[metric_name]
val_metric = history.history['val_' + metric_name]
e = range(1, EPOCHS + 1)
plt.plot(e, metric, 'bo', label='Train ' + metric_name)
plt.plot(e, val_metric, 'b', label='Validation ' +
metric_name)
plt.xlabel('Epoch number')
plt.ylabel(metric_name)
plt.title('Comparing training and validation ' +
metric_name + ' for '
+ model.name)
plt.legend()
plt.show()
```

3. Calling this function on the **`history`** object will show how the training loss and validation loss varies with each epoch:

```
import sys; sys.path.append('..')
from common.plot_util import eval_metric
eval_metric(model,history, 'loss',EPOCHS)
```

Figure 1.6 is the result of executing the preceding code:

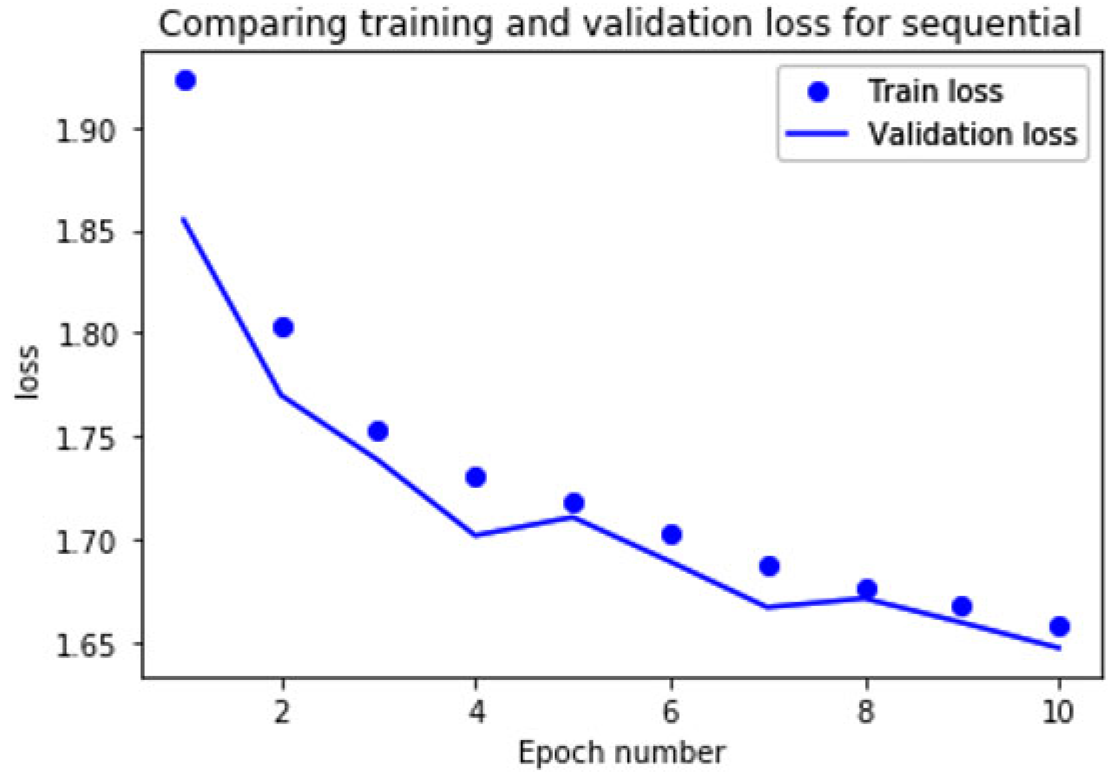


Figure 1.6: *Training loss and validation loss as a function of epochs*

While the training loss comes down gradually, the validation loss graph is not very smooth.

4. Next, let us also look at the training and validation accuracy as shown in *Figure 1.7*:

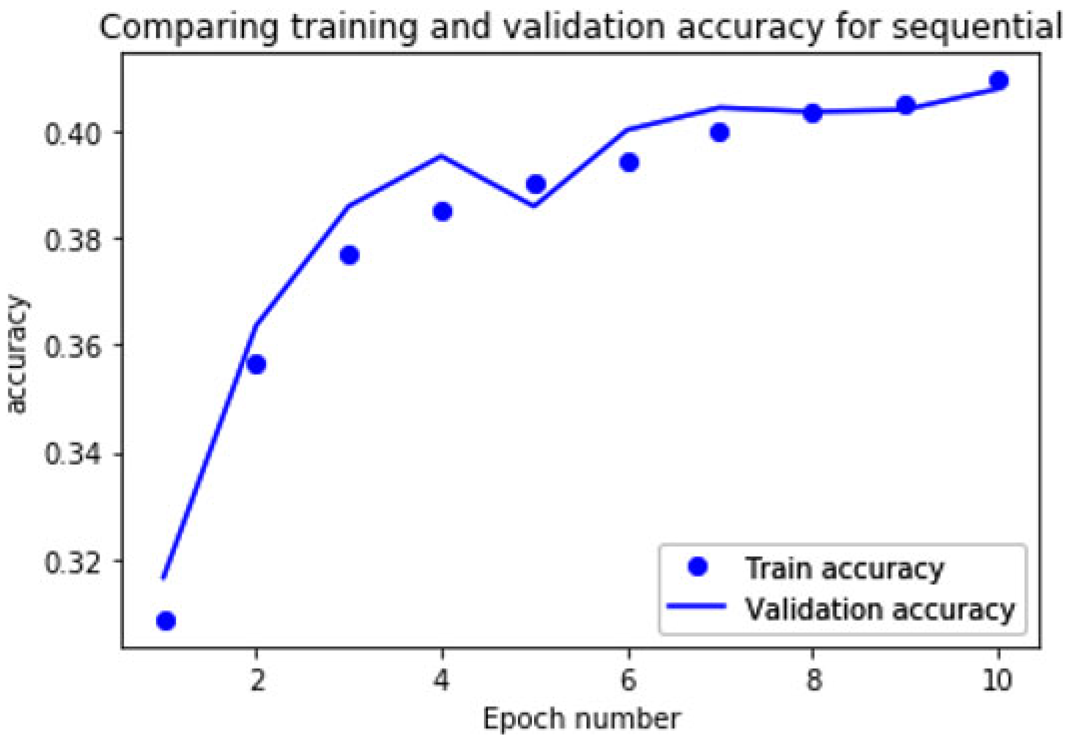


Figure 1.7: *Training accuracy and validation accuracy as a function of epochs*

While the training accuracy comes down gradually, the validation accuracy graph is not very smooth; the model's prediction varies quite a bit.

Verifying predictions

After training the model, you can make predictions on the same images. Let us look at the first image, predictions, and prediction array. Correct prediction labels are green, and incorrect predictions are red. The following number gives a percentage for the predicted label:

```
predictions = model.predict(test_images)
predictions[0]
```

Output array will be an array with **10 elements** that are probabilities of images belonging to a label:

```
array([0.05454378, 0.03618221, 0.11336125, 0.27892277,
0.06117007,
0.17112778, 0.14862633, 0.01250532, 0.11612879, 0.00743172],
dtype=float32)
```

You can see which label has the highest confidence value:

```
import numpy as np
```

```
np.argmax(predictions[0])
```

It will print output of **3,** which is the label of the image.

Let us draw the image and plot the value array. We will define two functions:

- The first one is **`plot_image(..)`**, which will show the image with the label printed following the image. The color of the label will be green if it is accurate, else it will be red.
- In the second function, **`plt.xlabel(…)`**, we will plot the array with the ‘true’ label as green and the ‘predicted’ label as red. If both are the same, only the green bar will show up:

```
def plot_image(i, predictions_array, true_label, img):
 predictions_array, img = predictions_array, img[i]
 true_label_local = true_label[i][0]
 plt.grid(False)
 plt.xticks([])
 plt.yticks([])
 plt.imshow(img, cmap=plt.cm.binary)
 predicted_label = np.argmax(predictions_array)
 if predicted_label == true_label_local:
 color = 'blue'
 else:
 color = 'red'
plt.xlabel("{} {:2.0f}%
({})".format(class_names[predicted_label],
       100*np.max(predictions_array)
       ,
       class_names[true_label_local]
       ),
       color=color)
def plot_value_array(i, predictions_array, true_label):
 true_label2 = true_label[i][0]
 plt.grid(False)
 plt.xticks(range(10))
 plt.yticks([])
 thisplot = plt.bar(range(10), predictions_array,
 color="#777777")
```

```
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)
    thisplot[predicted_label].set_color('red')
    thisplot[true_label2].set_color('green')
```

Calling these functions on **`predictions [0]`**:

```
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
```

Figure 1.8 shows the plot output for **`predictions[0]`**:

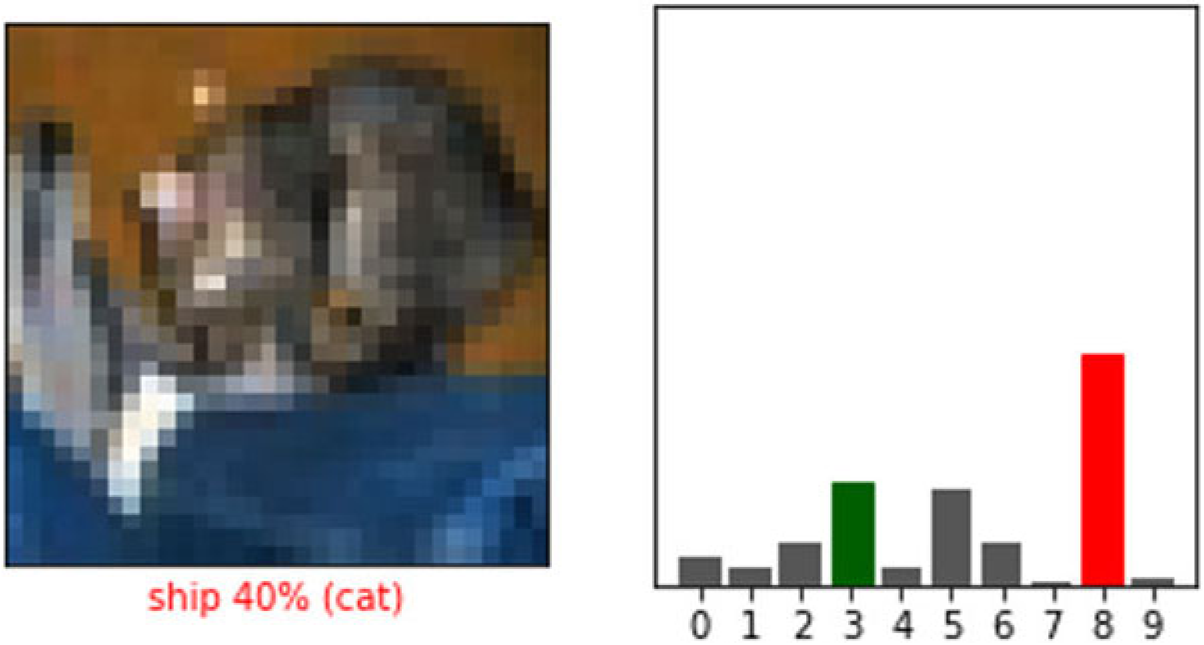


Figure 1.8: *Prediction versus actual for i=0 test image*

As you can see, the predicted label is **ship,** whereas the actual label is **cat**. Next, let us look at the **`i=5`** test image:

```
i = 5
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
```

```
plot_value_array(i, predictions[i], test_labels)
plt.show()
```

Figure 1.9 shows the plot output for **`predictions[5]`**:

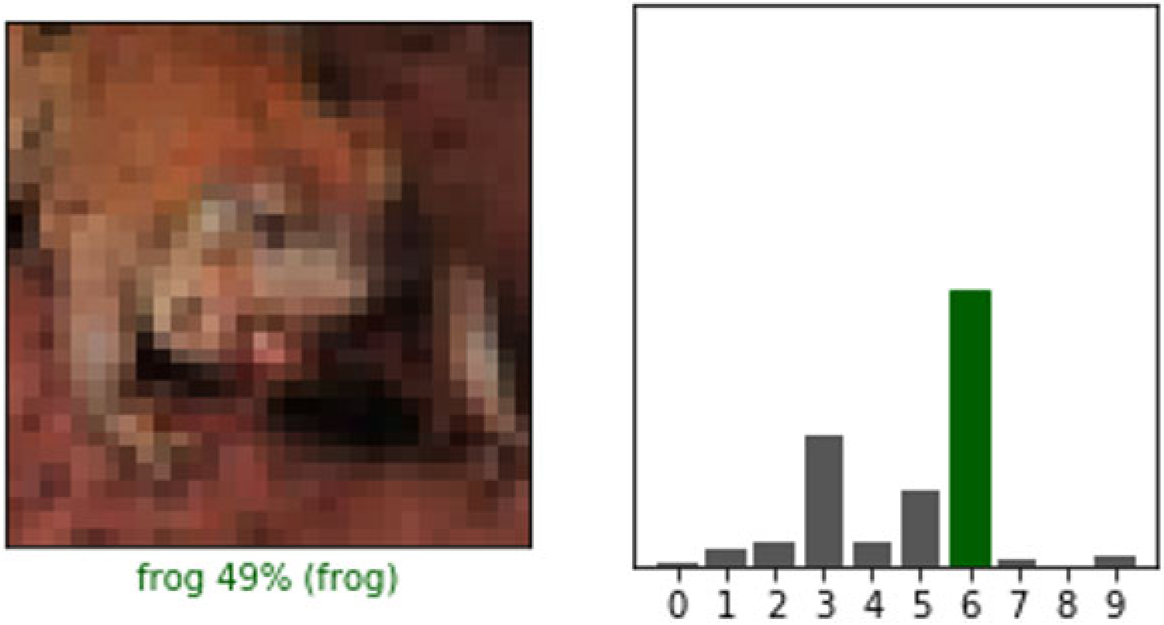


Figure 1.9: *Prediction versus actual for i=5 test image*

Low-level APIs

In this section, let us look at the low-level APIs and where they are they used. We will specifically look at the TensorFlow graph concept, and how it is leveraged.

Dataflow

Dataflow is a programming model used in parallel computing. The node represents units of computation in a dataflow graph. The edges are the data consumed or produced by the node. By representing computation using graphs, we have the ability to run forward or backward passes for training parameters of an ML model using algorithms like **gradient descent,** and applying chain rule to calculate the gradient at each node.

Advantages are as follows:

- parallelism

- computation optimization
- serialization using language neutral model
- distributed execution

Let us look at an example of a default graph:

```
# The function to be traced.
from datetime import datetime
@tf.function
def my_func(A,x,b):
 y = tf.add(tf.matmul(A, x), b, name="result")
 return y
# Set up logging.
stamp = datetime.now().strftime("%Y%m%d-%H%M%S")
logdir = './logs/func/%s' % stamp
writer = tf.summary.create_file_writer(logdir)
A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
x = tf.constant([[0, 10], [0, 0.5]])
b = tf.constant([[1, -1]], dtype=tf.float32)
# Bracket the function call with
# tf.summary.trace_on() and tf.summary.trace_export().
tf.summary.trace_on(graph=True, profiler=True)
# Call only one tf.function when tracing.
y = my_func(A,x,b)
with writer.as_default():
 tf.summary.trace_export(
  name="my_func_trace",
  step=0,
  profiler_outdir=logdir)
```

In these lines, there are a lot of details of TensorFlow, and its way of building a computational graph. This graph represents the matrix product between the constant tensor identified by the python variable and the constant tensor identified by the **`x`** Python variable, and the sum of the resulting matrix with the tensor identified by the **`b`** Python variable.

The result of the computation is represented by the **`y`** Python variable, also known as the output of the **`tf.add`** node, named result in the graph.

There is a separation between the concept of a Python variable and a node in the graph: we're using Python only to describe the graph; the name of the Python variable means nothing in the definition of the graph.

Moreover, we created **`writer = tf.summary.create_file_writer(logdir)`** to save a graphical representation of the graph we've built. The writer object has been created, specifying the path to store the representation (**`./log/matmul`**), and a trace on **`tf.Graph`** object obtained using the **`tf.summary.trace_on(graph=True, profiler=True)`** function call and then execute with **`writer.as_default(): tf.summary.trace_export(..)`**. *Figure 1.10* shows the Dataflow graph plotted using TensorBoard:

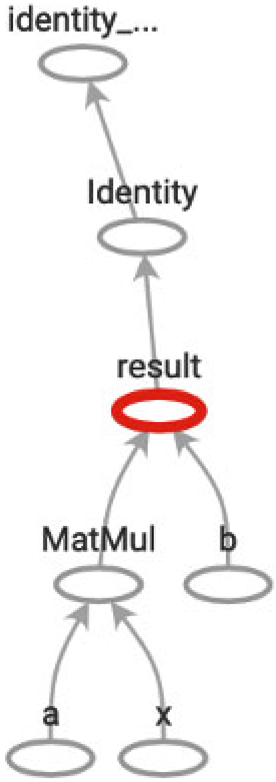
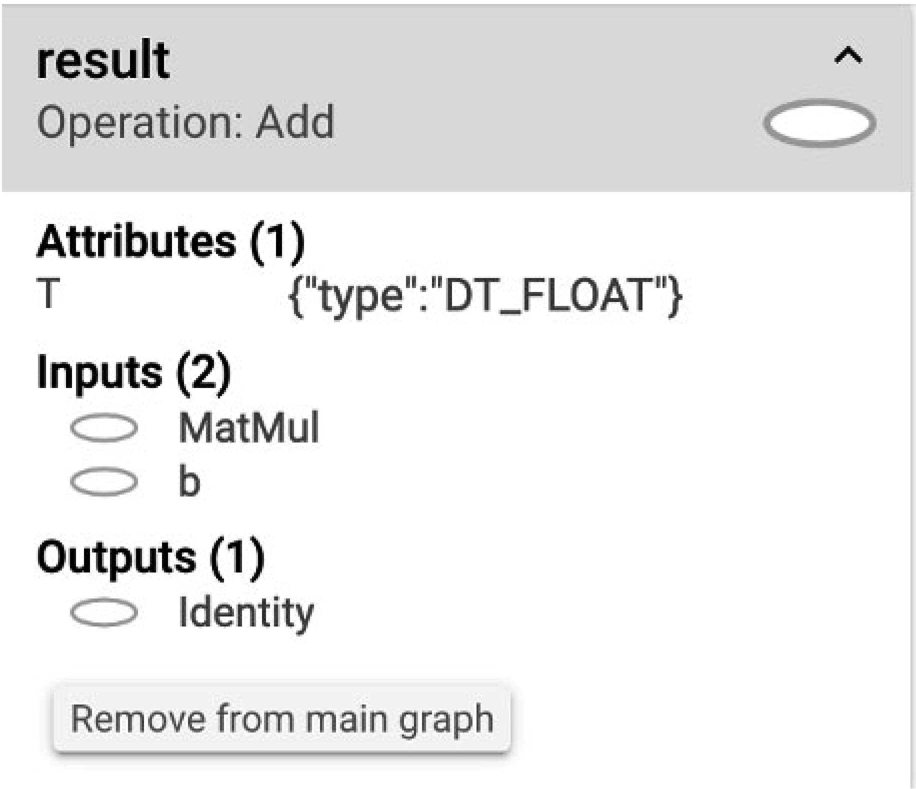


Figure 1.10: *Dataflow graph created by the TensorFlow sample*

Figure 1.11 shows the details of the result operation with **attributes**, **input**, and **output** tensors:



result
Operation: Add

Attributes (1)
T {"type":"DT_FLOAT"}

Inputs (2)
MatMul
b

Outputs (1)
Identity

Figure 1.11: *Dataflow graph created by the TensorFlow sample*

This is the computational graph that describes the operation *y* = *Ax* +*b*. The result node is highlighted in red in the preceding image and its details are shown in the right-hand column.

We are just creating and describing the graph—the calls to the TensorFlow API are just adding operations (nodes) and connections (edges) among them; there is no computation performed in this phase. In *TensorFlow 1.x*, the following approach needs to be followed—static graph definition and execution, while this is no longer mandatory in 2.x.

Since the computational graph is the fundamental building block of the framework (in every version), it is important to understand it in depth, since even after transitioning to 2.x, having an understanding of what's going on under the hood will help.

tf.Graph structure

As stated previously, there's no relation between the Python variables' names and the names of the nodes. TensorFlow is a C++ library, and we have used Python to build a graph in an easier way. The Python APIs in the

2.x version simplify the graph description phase, since they even create a graph implicitly, without the need to explicitly define it. There are two ways to define a graph:

- **Implicit**: It will define a graph using the `tf.*` methods. TensorFlow defines a default `tf.Graph`, accessible by calling `tf.get_default_graph`, if the graph is not explicitly defined. If the implicit definition is used, it limits the expressive power of a TensorFlow application, since it is constrained to using a single graph.
- **Explicit**: It is possible to define a computational graph explicitly and have more than one graph per application. This option has more expressive power, but is usually not needed since applications that need more than one graph are not common.

In order to explicitly define a graph, TensorFlow allows the creation of `tf.Graph` objects that, through the `as_default` method, create a context manager; every operation defined inside the context is placed inside the associated graph. A `tf.Graph` object defines a namespace for the `tf.Operation` objects it contains:

```
g = tf.Graph()
with g.as_default():
  # Define operations and tensors in `g`.
  c = tf.constant(30.0)
  assert c.graph is g
```

`tf.Graph` uses the collection mechanism to store metadata associated with the graph structure. A collection is uniquely identified by a key, and its content is a list of objects/operations. The user does not need to worry about the existence of a collection since they are used by TensorFlow itself to correctly define a graph.

You can use following the API to add a collection to the graph:

```
add_to_collection(
    name,
    value
)
```

When defining a parametric machine learning model, the graph must know which `tf.Variable` objects are the variables to update during the learning phase, and which other variables are not part of the model but are something else (such as moving the mean/variance computed during the training process—these are variables but not trainable). In this case, when, as we will see in the following section, a **`tf.Variable`** is created, it is added by default to two collections: the **global variable** and **trainable variable** collections.

tf.Operation and tf.Tensor

A Dataflow graph is a representation of computation, and nodes are units of computation. Each node is represented by **`tf.Operation`** and it has data flowing in and out through edges and is represented by **`tf.Tensor`**.

Referring to our example, when we call **`tf.constant([[1, 2], [3, 4]], dtype=tf.float32):`** a new node, (tf.Operation) named Const, is added to the default **`tf.Graph`**. This node returns a **`tf.Tensor (edge)`** named **`Const:0`**.

Each node in the TF graph is unique, in case there is a node already named **`Const`** in the graph (which is the default name given to all the constants), TensorFlow makes the node name unique by appending the suffix **`_1`**, **`_2`**. If a name is not provided, TensorFlow runtime gives a default name to each of the operation added, it also adds the suffix to make them unique in this case too.

Let us define two graphs and change the scope using **`tf.name_scope`** and print the variables:

```
import tensorflow as tf
graph1 = tf.Graph()
graph2 = tf.Graph()
with graph1.as_default():
   A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
   x = tf.constant([[0, 10], [0, 0.5]])
   b = tf.constant([[1, -1]], dtype=tf.float32)
   y = tf.add(tf.matmul(A, x), b, name="result")
with graph2.as_default():
   with tf.name_scope("scope_a"):
    x = tf.constant(1, name="x")
    print(x)
   with tf.name_scope("scope_b"):
    x = tf.constant(10, name="x")
    print(x)
   y = tf.constant(12)
   z = x * y
```

Output will show name scope specific variables:

```
Tensor("scope_a/x:0", shape=(), dtype=int32)
Tensor("scope_b/x:0", shape=(), dtype=int32)
```

In the last section of this chapter, let us focus on how TensorFlow 2.x does image classification.

Image classification with CNN

In this section, we look at how TensorFlow 2.x supports **Convolutional Neural Networking** (**CNN**) technique for image classification. We will look at CNNs in more detail in the later part of the book.

CNN overview

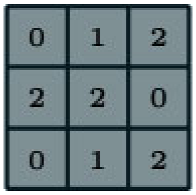
The bread and butter of neural networks are affine transformations: a vector is received as input. It is multiplied with a matrix to produce an output, and then, a bias vector is added before passing the result through a nonlinearity. This applies to any input: an image, a sound clip, or an unordered collection of features; whatever their dimensionality, their representation can always be flattened into a vector before the transformation.

Data like images, sound clips, and many other similar kinds of data have an intrinsic structure. More formally, they share these essential properties:

- These are stored as multi-dimensional arrays.
- The data features one or more axes, for which ordering matters (for example, width and height axes for an image, time axis for a sound clip, etc.).
- One axis, called the **channel axis**, is used to access different views of the data (for example, the red, green, and blue channels of a color image, or the left and right channels of a stereo audio track).

Properties listed previously are not exploited when an affine transformation (affine transformation is a linear mapping method that preserves points, straight lines, and planes) is applied. All the axes of the data elements are treated in the same way, and the topological information is not considered. Implicit structure of the data can be useful in solving some use cases, like, computer vision and speech recognition, and in these cases, it should be preserved.

Discrete convolutions come to the rescue there. A discrete convolution is a linear transformation that preserves this ordered information. It is sparse (a few input units contribute to an output unit) and heavily reuses parameters (same weights (kernel) is applied to multiple locations in the input). Figure 1.x illustrates an example of a discrete convolution. The light blue grid is the input feature map. For simplicity, a single input feature map is represented, but it is not uncommon to have multiple feature maps stacked onto one another. A kernel (shaded area) of value:



0	1	2
2	2	0
0	1	2

Figure 1.12: *Convolution filter*

Next, the figure shows the step-by-step process of applying this convolution to a 5x5 matrix:

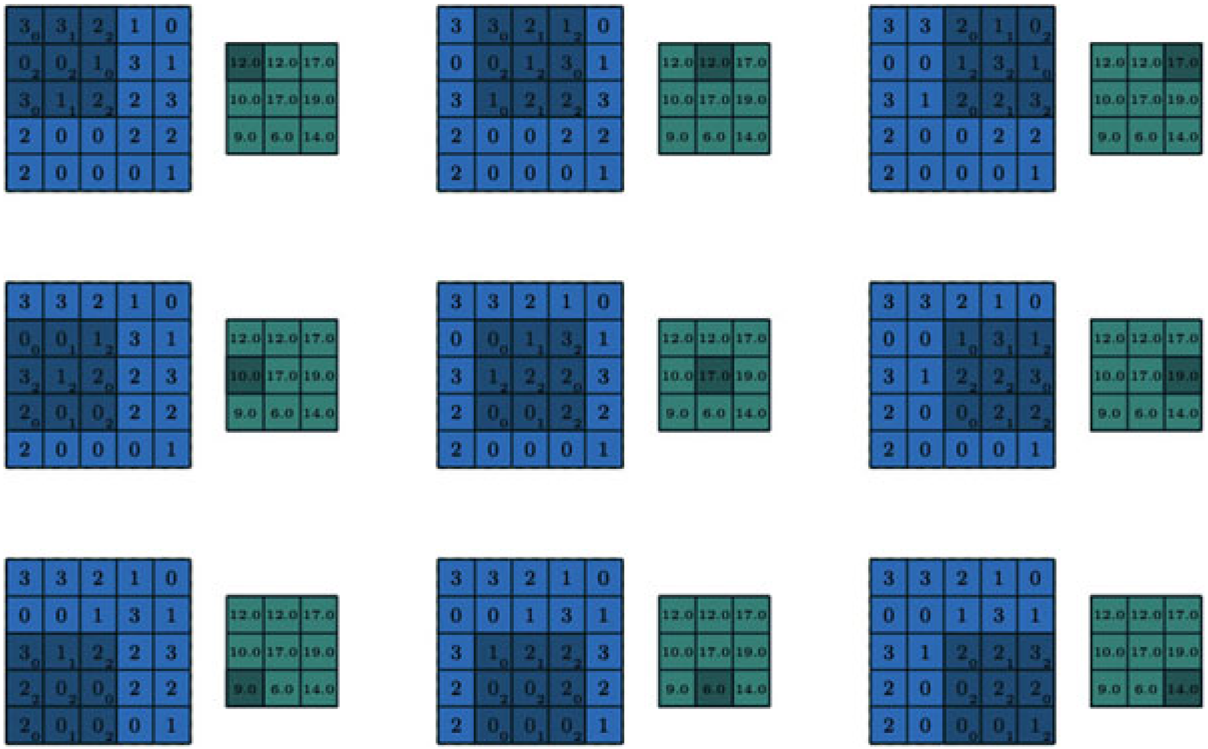


Figure 1.13: *Convolution filter applied to a 5 x 5 matrix resulting in a 3x3 matrix output*

Let us use the **cifar-10 dataset** we used earlier. Since the **cifar** dataset comes pre-packaged with TensorFlow, we will load the dataset:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()
```

The original dataset image is as follows, shown in *figure 1.14* for convenience:

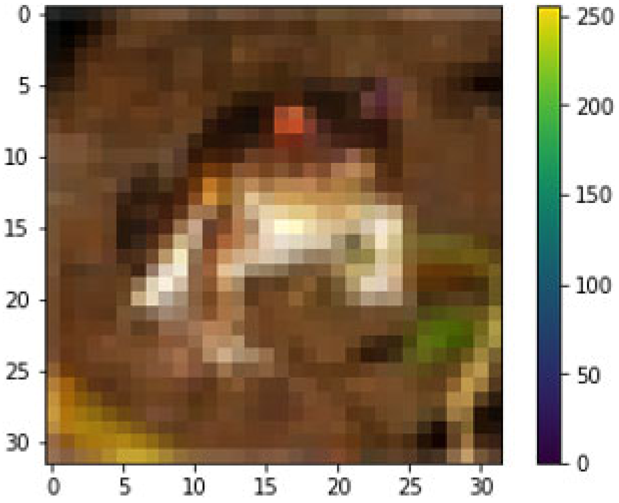


Figure 1.14: *Training image*

As we discussed earlier, the default images have pixel values ranging from *0* to *255*, these need to be normalized from *0* to *1*:

```
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_a
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```

On execution of the normalized samples, you will see the plot similar to *figure 1.15*:

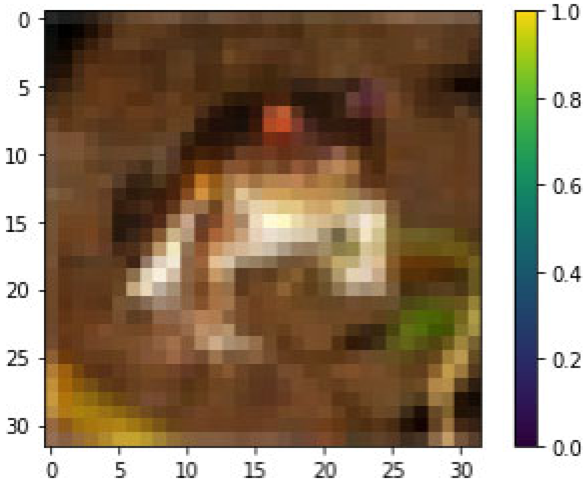


Figure 1.15: *Normalized training image*

We will create a sequential model with one convolutional layer. This layer is followed by a layer that flattens the tensor. Then, we will have two dense layers reducing the number of neurons to `64` and `10` respectively:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(32, 32, 3)))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

`layers.Conv2D`: This layer creates a kernel which convolves with the layer input to produce the output tensor. We are using `relu` activation, which is applied to the outputs.

Once the model has been created, let us train it:

```
model.compile(optimizer='adam',
      loss='sparse_categorical_crossentropy',
      metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=1,
        validation_data=(test_images, test_labels))
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 31s
612us/sample - loss: 1.6224 - accuracy: 0.4025 - val_loss:
1.4015 - val_accuracy: 0.4867
Epoch 2/10
50000/50000 [==============================] - 33s
653us/sample - loss: 1.3586 - accuracy: 0.5032 - val_loss:
1.3817 - val_accuracy: 0.4926
...
50000/50000 [==============================] - 44s
876us/sample - loss: 0.9779 - accuracy: 0.6505 - val_loss:
1.2214 - val_accuracy: 0.5756
Epoch 9/10
50000/50000 [==============================] - 43s
863us/sample - loss: 0.9238 - accuracy: 0.6693 - val_loss:
1.2404 - val_accuracy: 0.5756
Epoch 10/10
```

```
50000/50000 [==============================] - 43s
860us/sample - loss: 0.8740 - accuracy: 0.6895 - val_loss:
1.3040 - val_accuracy: 0.5658
```

As you can see, the validation accuracy of `0.56` is much better than the fully connected dense network we looked at earlier. Let us plot the training and validation loss as shown in the following figure:

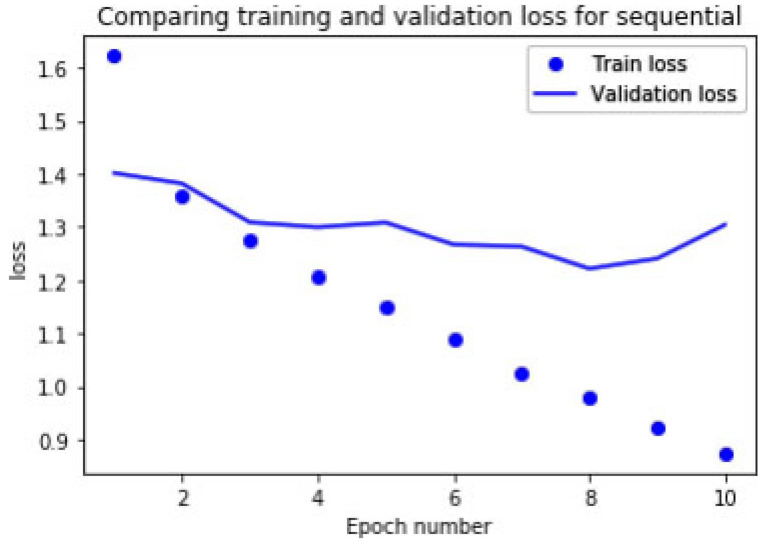


Figure 1.16: *Training and validation loss for a very basic CNN model*

The following figure shows the training and validation accuracy. Validation accuracy tapers off after reaching `0.56,` but the training accuracy is much better:

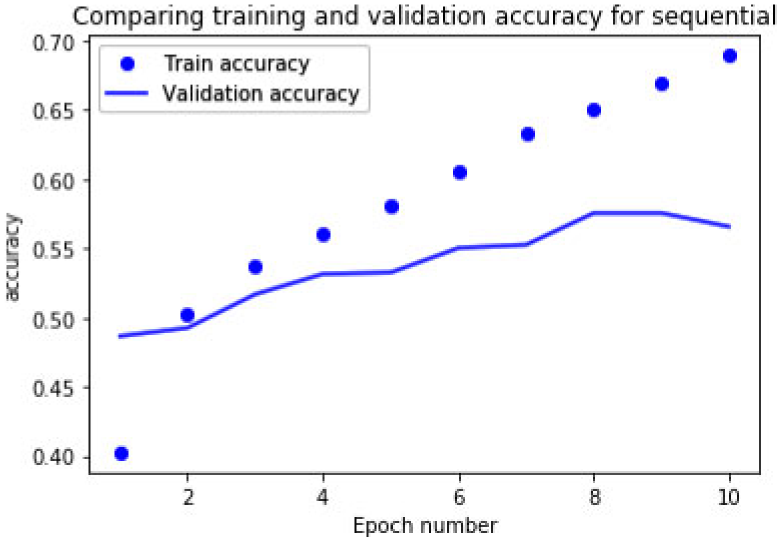


Figure 1.17: *Training and validation accuracy for a very basic CNN model*

The model is overfitting. We will look at techniques on how to fix this in the later chapters.

Clearly, the model is suboptimal and is not able to do a good job, since the accuracy tapers off after *10 epochs*.

Conclusion

In this chapter, we learnt how to get started with TensorFlow 2.x; from installation to writing your first program. We also looked at the building blocks like tensors, layers, and graphs with simple examples. We used **cifar-10 dataset** to do image classification using fully connected and **Convoluted Neural Network** topologies. These are some of the topics that we covered; we will go into more details in the subsequent chapters.

Questions

Use TensorFlow 2.x to predict admission to a graduate program. The dataset can be found clicking on the following link:

https://www.kaggle.com/mohansacharya/graduate-admissions.

References

- All symbols in TensorFlow 2: **https://www.tensorflow.org/api_docs/python**.
- A guide to Keras functional API: **https://www.perfectlyrandom.org/2019/06/24/a-guide-to-keras-functional-api/**.
- A guide to convolution arithmetic for deep learning: **https://arxiv.org/pdf/1603.07285.pdf**.

CHAPTER 2

Machine Learning with TensorFlow 2.x

Introduction

In this chapter, we are going to start by solving a regression problem with TensorFlow 2.x (**Housing Prices** dataset). We will explain the basics of regression; we will follow this with the first sample of predicting housing prices based on the features of the house being sold. Next, we will implement classification using the **Pima Indians** dataset to classify whether a person has diabetes or not. This dataset is explained in more detail in the section on classification. Then, we will cover the topic of handling underfitting and overfitting while training models. Last but not least, we will look at persisting and reloading the persisted models using stock TensorFlow 2.x APIs.

Structure

In this chapter, we will cover the following topics:

- Basic classification with TensorFlow 2.x
    - Clean the dataset
    - Model creation and training
- Regression examples with TensorFlow 2.x
    - The Boston Housing Price dataset
    - Preparing the data
    - Validating our approach using K-fold validation
- Overfitting and underfitting
- Saving and restoring models

Objectives

In this chapter, we will learn how to use TensorFlow 2.x to solve regression and classification problems. We will be using fully connected neural networks to solve these problems, instead of traditional machine learning algorithms. We will tackle the issue of underfitting and overfitting using techniques like **L1** and **L2 norm** as well as applying dropout. We will also look at saving and restoring the models using the new APIs in version 2.x.

Basic classification with TensorFlow 2.x

Let us start by exploring the **Pima Indians** dataset for diabetes detection. This dataset was sourced from *National Institute of Diabetes and Digestive and Kidney Diseases*.

It is aimed at providing data to diagnostically predict whether a patient has diabetes or not. Constraints were placed on selection of the dataset from the larger database. All the patients were females, 21 years of age, and of **Pima Indian heritage**.

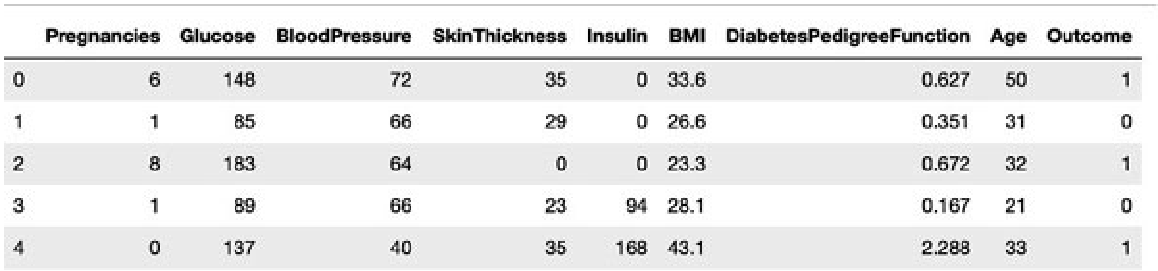
In our sample, we will create a three-layer neural network and learn about model check-pointing and saving. In the end, we will plot the training history. Let us start by importing the relevant classes from **`numpy`**, **`pandas`**, and **`TensorFlow`**:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g.
pd.read_csv)
from subprocess import check_output
import matplotlib.pyplot as plt
%matplotlib inline
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import ModelCheckpoint
seed = 42
np.random.seed(seed)
```

Next, we will load the dataset from a **`csv`** file using **`pandas`**:

```
pdata = pd.read_csv('../data/diabetes.csv')
#the dataset can be analyzed using the standard head() command
pdata.head()
```

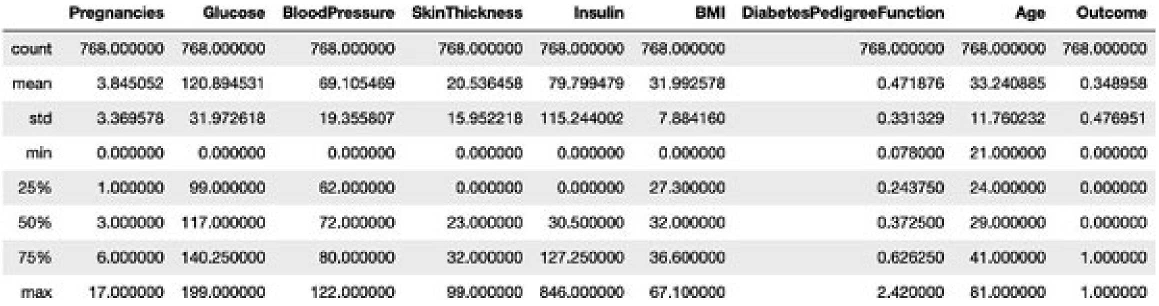
Figure 2.1 shows the first five sample lines using the **`head()`** command on **`pandas`** data frame:



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Figure 2.1: *Diabetes dataset sample lines*

Let us also look at the overall structure of the dataset using **`describe()`** function as shown in *figure 2.2*:



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Figure 2.2: *Summary table shows how various features vary*

This is a good way to summarize data in a tabular format for the dataset. Let us look at how some variables interplay with each other.

We will use the **`seaborn`** library for plotting this data:

```
import seaborn as sns
import pandas as pd
sns.set()
diabetes = pd.read_csv('../data/diabetes.csv')
g = sns.pairplot(diabetes,height=3,
       vars=["Age", "BMI", "Pregnancies"],hue = 'Outcome')
```

The output of the preceding code will be a *3x3* plot with three variables (**`Age`**, **`BMI`**, **`Pregnancies`**) in pairs, and differentiated by whether the outcome was **`Pregnancies(1)`** or not (0), as shown in the *figure 2.3*:

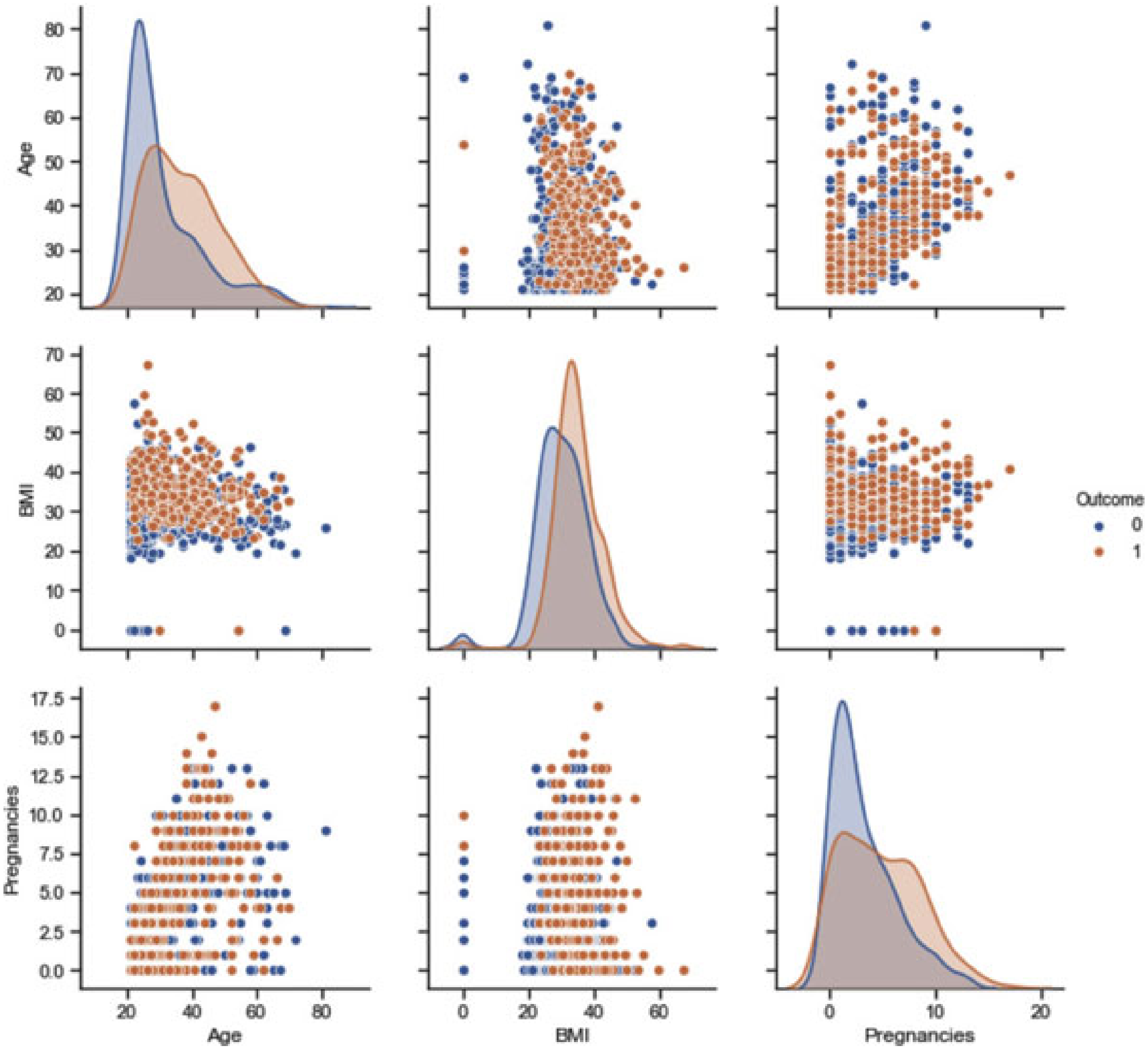


***Figure 2.3:** Pair-plot between three variables for Pima Indian dataset*

- **Interpretation**: If the BMI is higher, the probability or outcome is higher as well. There is little correlation between **`Age`** and **`BMI`**. Let us plot the regression plots to understand more:

```
g = sns.pairplot(diabetes,height=3,
     plot_kws={ 'scatter_kws': {'alpha': 0.1}},
     vars=["Age", "BMI","Pregnancies"],kind="reg",hue =
     'Outcome')
```

The output will again be a *3x3* pair-plot, but with regression lines as shown in the *figure 2.4*:

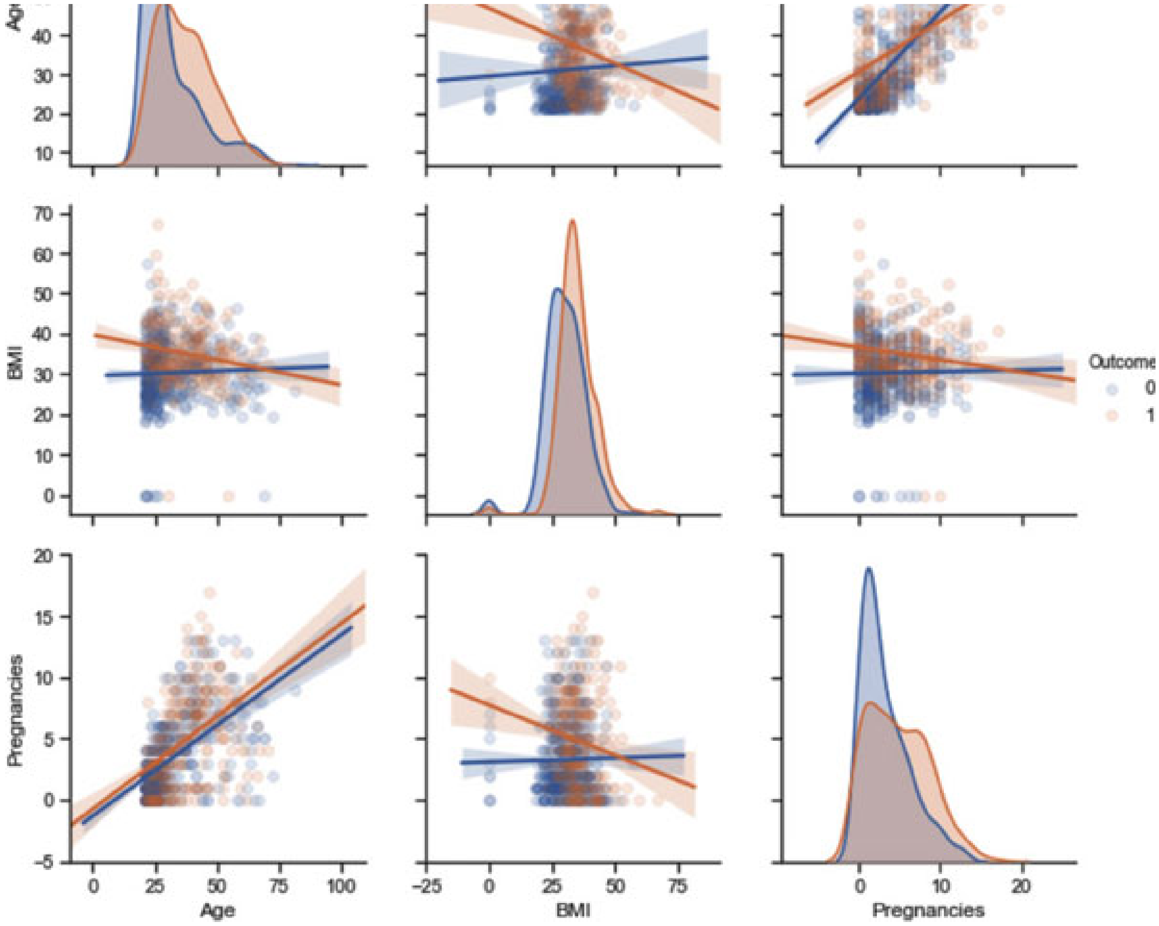


Figure 2.4: Pair-plot with regression lines

- **Age and BMI**: There is negative correlation between `Age` and `BMI` for `Outcome` of `1` and positive for `Outcome` of `0`.
- **Age and Pregnancies**: There is a higher positive correlation if the `Outcome` is `0`.

Clean the dataset

Before moving forward, we need to inspect and clean the dataset.

1. Let's remove the `0-entries` for these fields listed previously.

   This is an optional step which is needed to make sure the dataset is clean.

```
zero_fields = ['Glucose', 'BloodPressure', 'SkinThickness',
'Insulin', 'BMI']
def check_zero_entries(data, fields):
 for field in fields:
  print('field %s: num 0-entries: %d' % (field,
    len(data.loc[ data[field] == 0,
    field ])))
```

The preceding code will help find the **`0-entries`** in the dataset:

```
field Glucose: num 0-entries: 5
field BloodPressure: num 0-entries: 35
field SkinThickness: num 0-entries: 227
field Insulin: num 0-entries: 374
field BMI: num 0-entries: 11
```

2. Next, split into train and test using **`sklearn.model_selection.train_test_split`**:

```
from sklearn.model_selection import train_test_split
features = list(pdata.columns.values)
features.remove('Outcome')
print(features)
X = pdata[features]
y = pdata['Outcome']
X_train, X_test, y_train, y_test = train_test_split(X, y,
 test_size=0.25, random_state=0)
print(X_train.shape)
```

The following output shows a list of features and train and test data-shape:

```
['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
 'Insulin', 'BMI', 'DiabetesPedigreeFunction',
'Age']
(576, 8)
(192, 8)
```

3. Create a function to impute zero fields to mean value:

```
def impute_zero_field(data, field):
  nonzero_vals = data.loc[data[field] != 0, field]
  avg = np.sum(nonzero_vals) / len(nonzero_vals)
  k = len(data.loc[ data[field] == 0, field]) # num of 0-
  entries
```

```
    data.loc[ data[field] == 0, field ] = avg
    print('Field: %s; fixed %d entries with value: %.3f' %
    (field, k, avg))
```

4. Next, apply this function and fix the training set:

```
for field in zero_fields:
    impute_zero_field(X_train, field)
# Fix for Test dataset
for field in zero_fields:
    impute_zero_field(X_test, field)
```

5. Remove the field-name labels:

```
X_train = X_train.values
y_train = y_train.values
X_test = X_test.values
y_test = y_test.values
```

Now, our training and test data is ready for model creation and training.

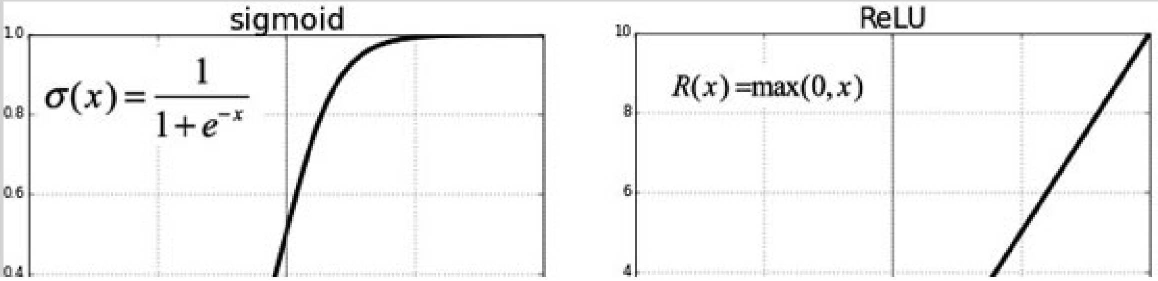
Model creation and training

Next, we will create the neural network. We will create a neural network with three layers. We are choosing three layers more as a starting point to keep it simple. Notice that the `input_dim` is `8` which is equal to the number of features in the dataset.

We are using **Rectified Linear Unit** (**ReLU**) activation in the first two layers and sigmoid in the last layer.

Note: ReLU () activation function: ReLU is the most popular activation function right now. This function is used in almost all the convolutional neural networks implementations.As you can see in the *figure 2.5*, the ReLU is half rectified (from bottom). *f(x)* is zero when *x* is less than zero and *f(x)* is equal to *x* when *x* is above or equal to zero.

Range of this function: *[0 to infinity)*:



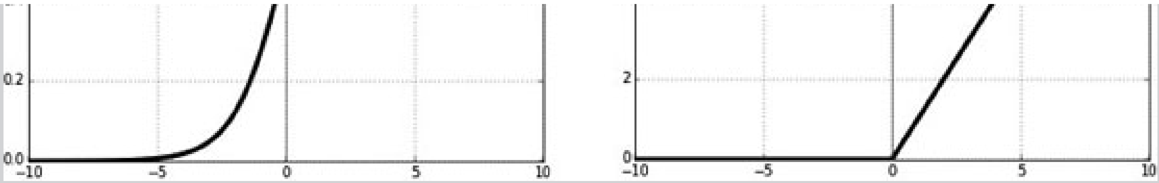


Figure 2.5: *Sigmoid versus ReLU activation function*

The ReLU function and its derivative both are **monotonic** (function and its first derivative (which need not be continuous) does not change sign.

Our model is a sequential model with three dense layers. A sequential model is a linear stack of layers and extends `training.model`.

The *figure 2.6* shows the class hierarchy followed by the sequential model:

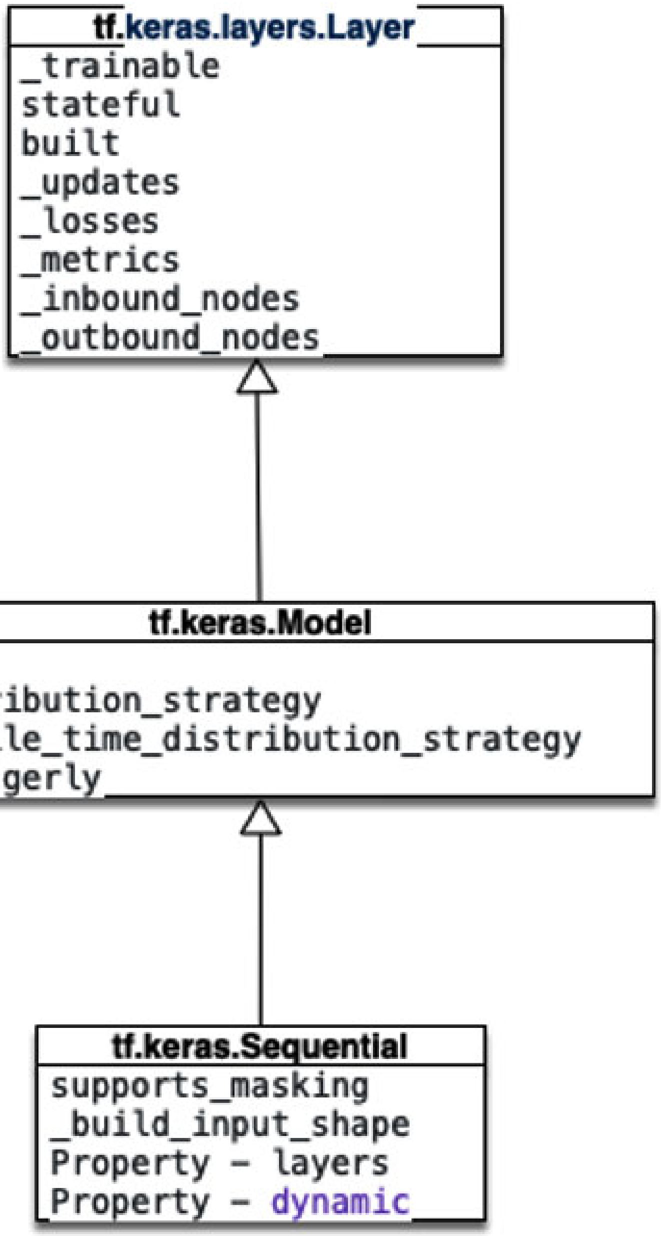


Figure 2.6: *Class hierarchy followed by the sequential*

It is a subclass of the **`tf.keras.Model`** class which subclasses **`tf.keras.layers.Layer`**.

All the models track input and output nodes, losses, and metrics. It also keeps track of whether the model is trainable and has been built already:

```
EPOCHS = 1000 # num of epochs to test for
BATCH_SIZE = 16
## Create our model
model = Sequential()
```

```
# 1st layer: input_dim=8, 12 nodes, RELU
#model.add(Dense(12, input_dim=8, init='uniform',
activation='relu'))
model.add(Dense(12, input_dim=8, activation='relu'))
# 2nd layer: 8 nodes, RELU
model.add(Dense(8, activation='relu'))
# output layer: dim=1, activation sigmoid
model.add(Dense(1, activation='sigmoid' ))
```

Note: Creating a sequential model: There are two ways of creating a sequential model:

```
from tensorflow. keras.models import Sequential
from keras.layers import Dense, Activation
model = Sequential([
   Dense(32, input_shape=(784,)),
])
```

You can also simply add layers via the .add() method:

```
model = Sequential()
model.add(Dense(32, input_dim=784))
```

We are using the second technique here.

Next, we compile the model and create a checkpoint:

```
# Compile the model
model.compile(loss='binary_crossentropy',
# since we are predicting 0/1
       optimizer='adam',
       metrics=['accuracy'])
# checkpoint: store the best model
ckpt_model = 'pima-weights.best.hdf5'
checkpoint = ModelCheckpoint(ckpt_model,
        monitor='val_accuracy',
        verbose=1,
        save_best_only=True,
        mode='max')
callbacks_list = [checkpoint]
```

Train the model and store the training iteration values in the history **`object.fit()`**. The function in **`tf.keras.train`** has the following signature:

```
fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose=1,
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_freq=1,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
    **kwargs
)
```

We will set some of the parameters as default, but set the following ones from the dataset as well as a few others hyper parameters:

- **`x`**: **`X_train`**
- **`y`**: **`y_train`**
- **`validation_data=(X_test, y_test)`**
- **`epochs=epochs`** variable we defined at the beginning to **`1000`**
- **`batch_size`**=**`16`**
- **`callbacks=callbacks_list`**

The preceding figure shows what happens when the **`model.fit`** call occurs. Behind the scene, a training loop is started depending on the configuration chosen.

Let us focus on getting the model trained:

```
history = model.fit(X_train,
        y_train,
        validation_data=(X_test, y_test),
        epochs=EPOCHS,
        batch_size=BATCH_SIZE,
        callbacks=callbacks_list,
        verbose=0)
```

Output of the **`model.fit`** will be similar to the following:

```
Starting training:
Epoch 00001: val_accuracy improved from -inf to 0.67708
...
Epoch 00999: val_accuracy did not improve from 0.79688
Epoch 01000: val_accuracy did not improve from 0.79688
```

Let us plot the accuracy and loss metrics for the training and validation data using **`matplotlib`**:

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```

Now, let us plot the loss and validation loss:

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'])
plt.show()
```

The output of the preceding code will produce a graph with **accuracy** and **validation accuracy**: **`val_accuracy`**, as a function of epochs, as shown in *figure 2.7*:

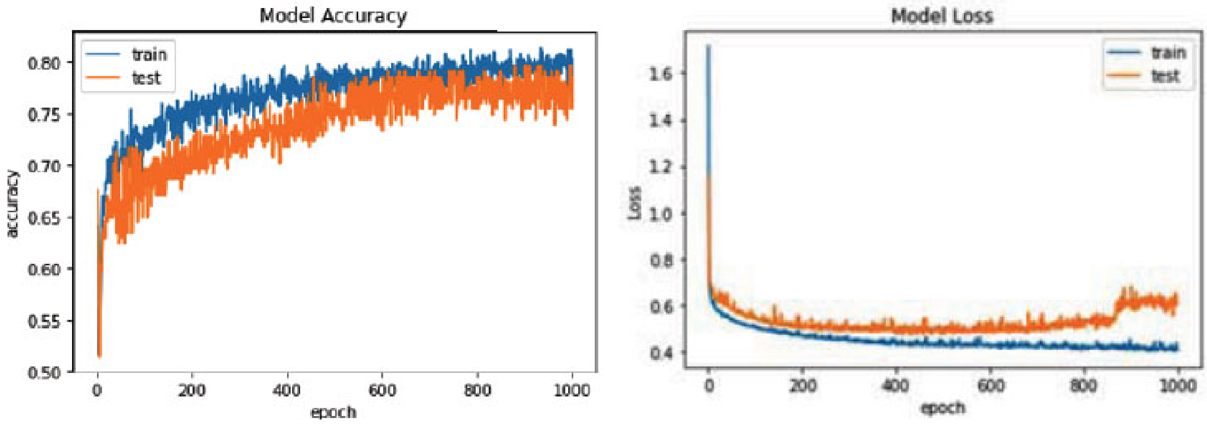


Figure 2.7: Model accuracy and loss of the model

Validation accuracy is lower than training accuracy, but by a lower margin. Losses for the test don't taper at the same rate as for training. This typically happens because of overfitting. Let us look at the model loss a little more carefully by ignoring the *first 10 entries*. *Figure 2.8* provides a plot of epoch versus loss from **10** to **1000** epochs:

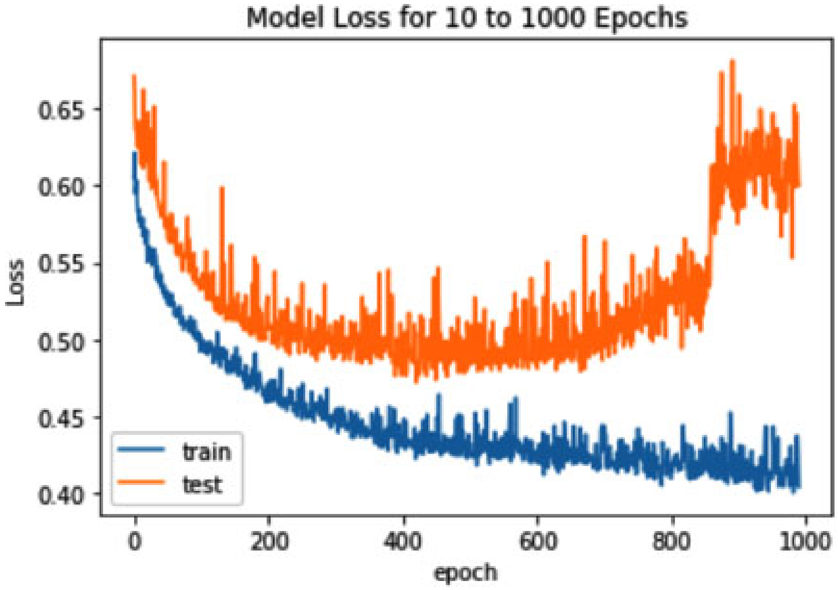


Figure 2.8: Model loss for epochs 10 to 1000

Overfitting is substantially more evident in this plot. The loss starts increasing after **800** epochs. Later in the book, we will look at some of the techniques on how this can be improved.

In the next section, we will look at a simple regression example with the TensorFlow 2.x stack.

Regression examples with TensorFlow 2.x

In the previous examples, we have considered classification problems where we want to predict a single discrete label for input data points. There is another category, of machine learning problem, called **regression,** where we predict a continuous value. For example, we can predict stock prices or insulin levels in a body from past data.

The Boston Housing Price dataset

We will be predicting the median price of homes in a given suburb of Boston in the mid-1970s based on data points like **crime rate**, **local property tax**, and so on. The dataset has very few data points: *506 plots into training samples of 404* and *test samples of 102*. Also, each feature has a different scale. Some values are between *0* and *1,* while others vary between *0* and *100* or *1* and *12*.

Let's take a look at the data:

```
from tensorflow.keras.datasets import boston_housing
(train_data, train_targets), (test_data, test_targets) =
boston_housing.load_data()
```

If we analyze the shape, you can see we have **13** features. The training data shape is:

```
train_data.shape
(404, 13)
```

The test data shape is:

```
test_data.shape
(102, 13)
```

There are `404` training samples and `102` test samples. The data is made up of `13` features. The `13` features in the input data are described as follows:

- per capita crime rate
- proportion of residential land zoned for lots over 25,000 square feet
- proportion of non-retail business acres per town
- Charles `river` variable (`1` if tract bounds river; `0` otherwise)
- nitric oxides concentration (parts per 10 million)
- average number of rooms per dwelling
- proportion of owner-occupied units built before *1940*
- weighted distances to five Boston employment centers

- index of accessibility to radial highways
- full-value property-tax rate per _$10,000_
- pupil-teacher ratio by town (_1000 * (B - 0.63) ** 2_ where _B_ is the proportion of Black people by town. _%_ lower status of the population)

The targets are the median values of owner-occupied homes, in thousands of dollars:

```
train_targets array([15.2, 42.3, 50. , 21.1, 17.7, 18.5, 11.3,
15.6, 15.6, 14.4, 12.1, 17.9, 23.1, 19.9, 15.7, 8.8, 50. , 22.5,
24.1, 27.5, 10.9, 30.8, 32.9, 24. , 18.5, 13.3, 22.9, 34.7, 16.6,
17.5, 22.3, 16.1, 14.9, ....., 28.7, 37.2, 22.6, 16.4, 25. ,
29.8, 22.1, 17.4, 18.1, 30.3, 17.5, 24.7, 12.6, 26.5, 28.7, 13.3,
10.4, 24.4, 23. , 20. , 17.8, 7. , 11.8, 24.4, 13.8, 19.4, 25.2,
19.4, 19.4, 29.1])
```

Output has been truncated.

Let us plot the training target to get a better idea of how the prices are varying across the training target:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(train_targets)
plt.title('Housing Prices Training Target')
plt.xlabel('Sample No')
plt.ylabel('Price')
```

The following plot makes it very clear that housing prices are varying quite a bit for _400+_ samples:

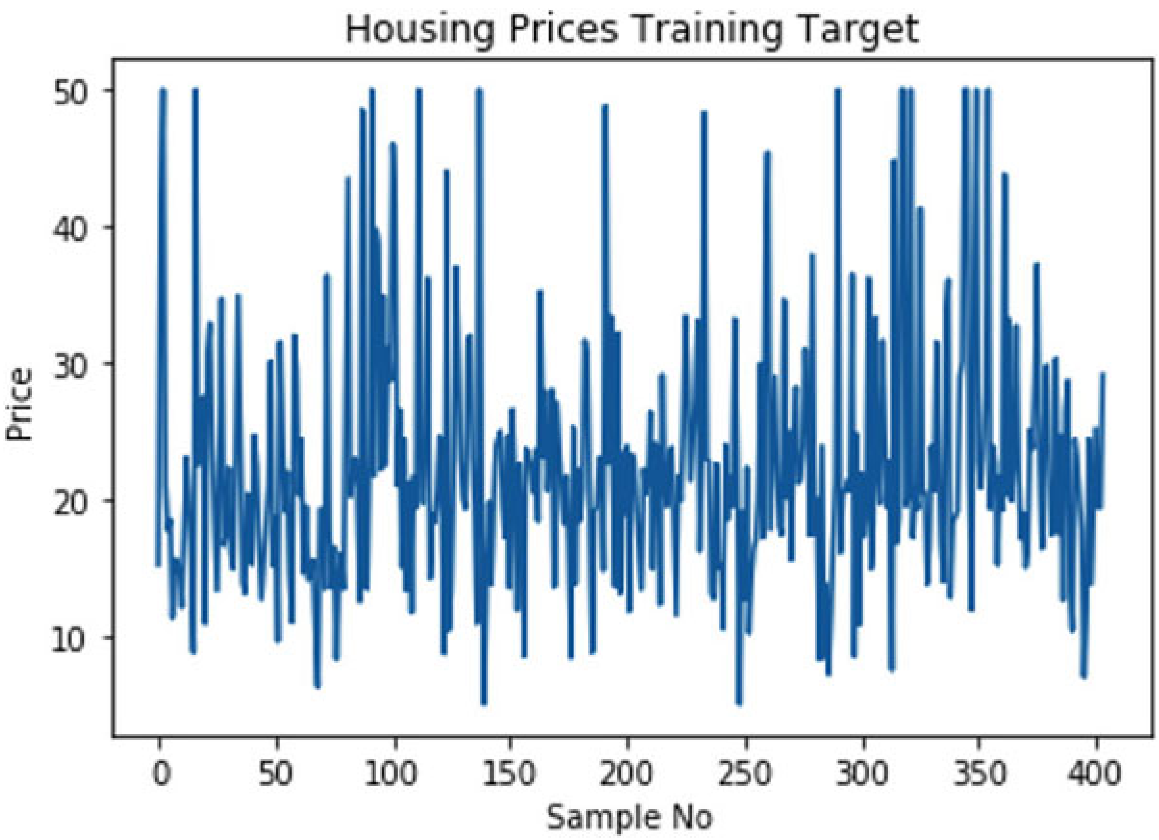


***Figure 2.9:** Housing prices training target*

Preparing the data

It will be difficult to feed the neural network values that take different ranges. The best practice is to do feature-wise normalization. This can be easily done in **NumPy**:

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

Please note the quantities (mean and standard deviation) we use for normalizing the test data from training data. We should never use any quantity computed on test data even for normalization.

Building the neural network

Since very few samples are available, we will use a small network with two hidden layers of `64` units each. Generally, if the training data is small, there will be overfitting. A smaller network is one way to mitigate this:

```
from tensorFlow.keras import models
from tensorFlow.keras import layers
def build_model():
    # Because we will need to instantiate
    # the same model multiple times,
    # we use a function to construct it.
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
            input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=
    ['mae'])
    return model
```

Our network's last layer is a single unit with no activation. This is required since it is a setup for scalar regression. Applying activation would constrain the range the output can take. If we apply a sigmoid activation, the network will learn to predict values between `0` and `1`, which is not what we want in this case.

Note that we have compiled the network with the **MSE** loss function: **Mean Squared Error**. It is calculated using the *square of the difference between the predictions and the targets*, a well-known loss function for regression problems.

We are also monitoring a metric during training called **MAE**. It stands for **Mean Absolute Error**. It is the *absolute value of the difference between the predictions and the targets*. For example, an MAE of *0.5* on this problem would mean that our predictions are off by *$500* on average.

Validating our approach using K-fold validation

To evaluate the network while we adjust the parameters, we could have split the dataset into training and validation set, as done in previous examples. But with such a small dataset, validation set work is very small. Validation scores would change a lot depending on the values that end up in the training or validation dataset. This will prevent us from reliable evaluation of the model.

In such cases, the best practice is to use **K-fold cross-validation**. It splits the available data into *K partitions* and then, instantiates *K-identical models*. These models are trained on *k-1 partitions* while evaluating them on the remaining partition. The validation score is calculated as an average of K-validation scores:

```
import numpy as np
k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print('processing fold #', i)
    # Prepare the validation data: data from partition # k
    val_data = train_data[i * num_val_samples: (i + 1) *
    num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) *
    num_val_samples]
    # Prepare the training data: data from all other partitions
    partial_train_data = np.concatenate(
      [train_data[:i * num_val_samples],
      train_data[(i + 1) * num_val_samples:]],
      axis=0)
    partial_train_targets = np.concatenate(
      [train_targets[:i * num_val_samples],
      train_targets[(i + 1) * num_val_samples:]],
      axis=0)
    # Build the Keras model (already compiled)
    model = build_model()
    # Train the model (in silent mode, verbose=0)
    model.fit(partial_train_data, partial_train_targets,
        epochs=num_epochs, batch_size=1, verbose=0)
    # Evaluate the model on the validation data
    val_mse, val_mae = model.evaluate(val_data, val_targets,
    verbose=0)
    all_scores.append(val_mae)
```

If you print **`all_scores`**, you will get a much better handle:

```
[2.3030572, 2.2186685, 2.6573195, 2.455313]
```

Let us calculate the mean **`np.mean(all_scores)`** which comes to **`2.4085896`**.

Different runs show rather different validation scores, from *2.1* to *2.9*. The average score of 2.4 is a much more reliable metric than any single of these scores--that is the thought process behind the K-fold cross-validation. In this use case, we are off by *2,400* on the average, which is still significant given that prices range from $*10,000* to *$50,000*.

Let us train the neural network for **`500`** epochs. We will modify the training loop to save the *per-epoch* validation score log to keep record of model performance at each epoch:

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
   print('processing fold #', i)
   # Prepare the validation data: data from partition # k
   val_data = train_data[i * num_val_samples: (i + 1) *
   num_val_samples]
   val_targets = train_targets[i * num_val_samples: (i + 1) *
   num_val_ samples]
   # Prepare the training data: data from all other partitions
   partial_train_data = np.concatenate(
    [train_data[:i * num_val_samples],
    train_data[(i + 1) * num_val_samples:]],
    axis=0)
   partial_train_targets = np.concatenate(
    [train_targets[:i * num_val_samples],
    train_targets[(i + 1) * num_val_samples:]],
    axis=0)
   # Build the Keras model (already compiled)
   model = build_model()
   # Train the model (in silent mode, verbose=0)
   history = model.fit(partial_train_data, partial_train_targets,
        validation_data=(val_data, val_targets),
        epochs=num_epochs, batch_size=1, verbose=0)
   #print(history.history)
   mae_history = history.history['val_mae']
   all_mae_histories.append(mae_history)
```

Let us first create predictions on **`test_data`** and compare with **`test_features`**:

```
predictions = model.predict(test_data)
```

And then, plot these predictions against the **`test`** targets:

```
import numpy as np
import matplotlib.pyplot as plt
# Create plots with pre-defined labels.
fig, ax = plt.subplots()
ax.plot(predictions, label='Predictions')
ax.plot(test_targets, label='Test Targets')
#legend = ax.legend(loc='upper center', shadow=False,
fontsize='x-large')
legend = ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left',
borderaxespad=0.)
plt.show()
```

The output of the plot will be similar to *figure 2.10*, showing predictions versus actual values from test targets:

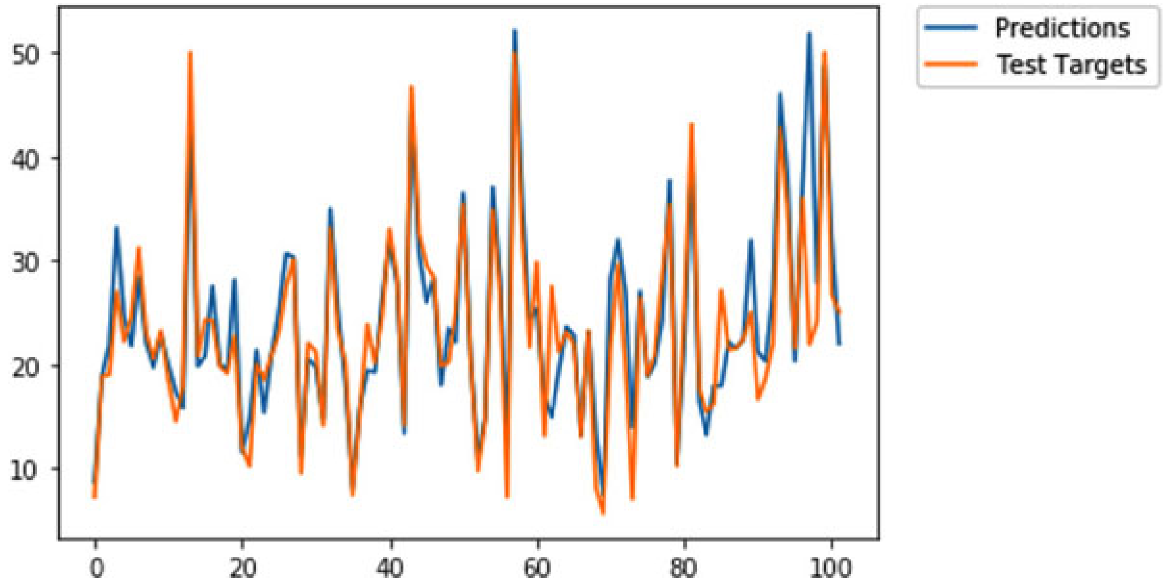


Figure 2.10: *Predictions versus test targets*

The plot shows that predictions are quite close to the actual value.

The next step is to check the MAE scores. We can then compute the average of the *per-epoch MAE* scores for all folds:

```
average_mae_history = [
   np.mean([x[i] for x in all_mae_histories]) for i in
   range(num_epochs)]
```

Let us plot the **`average_mae_history`** calculated previously:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
plt.plot(range(1, len(average_mae_history) + 1),
average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

Once the plot is created (refer to *figure 2.11*), you will notice how the MAE reduces at around *50* epochs and rises again:

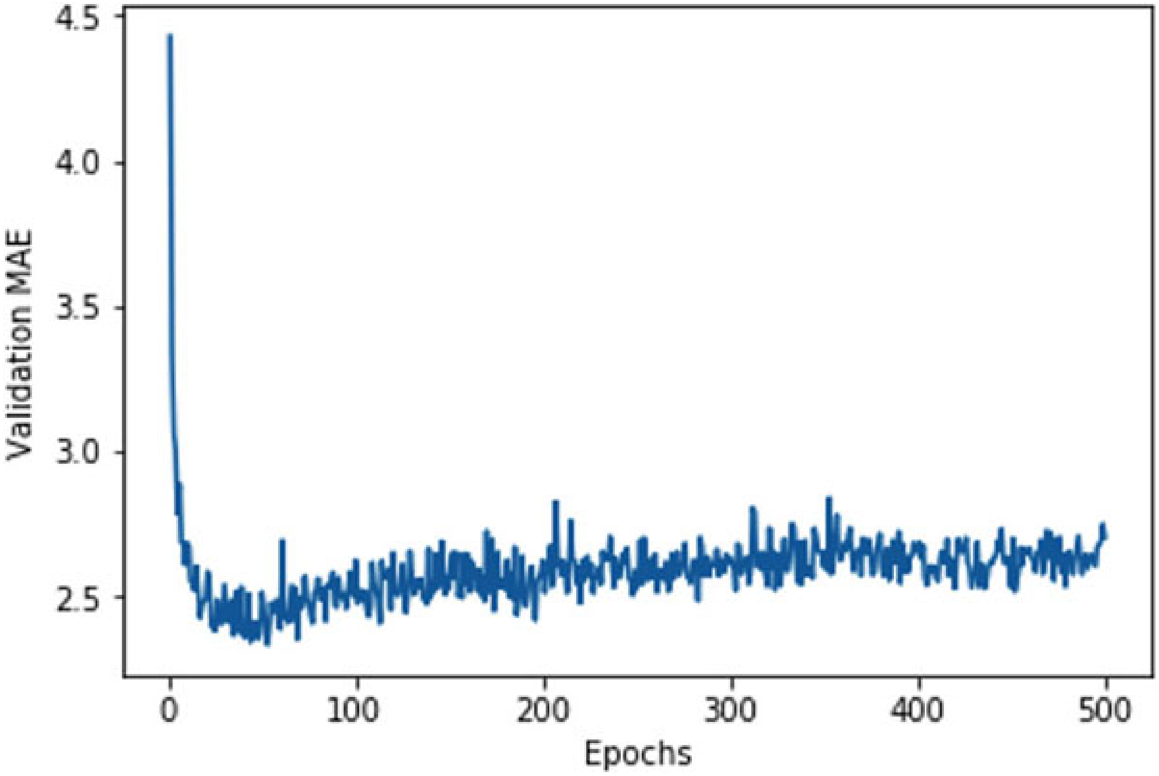


Figure 2.11: *Average MAE as a function of epochs*

It may be hard to see the plot due to scaling issues and relatively high variance. Let's omit the first `10` data points, which are on a different scale from the curve. We will replace each point with an exponential moving average of the previous points, to obtain a smooth plot.

```
plt.plot(range(1, len(average_mae_history[10:]) + 1),
average_mae_history[10:])
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
def smooth_curve(points, factor=0.9):
  smoothed_points = []
 for point in points:
  if smoothed_points:
  previous = smoothed_points[-1]
  smoothed_points.append(previous * factor + point * (1 -
  factor))
  else:
  smoothed_points.append(point)
 return smoothed_points
smooth_mae_history = smooth_curve(average_mae_history[10:])
plt.plot(range(1, len(smooth_mae_history) + 1),
smooth_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```

The validation MAE calculated previously is plotted as a function of epochs in *figure 2.12*:

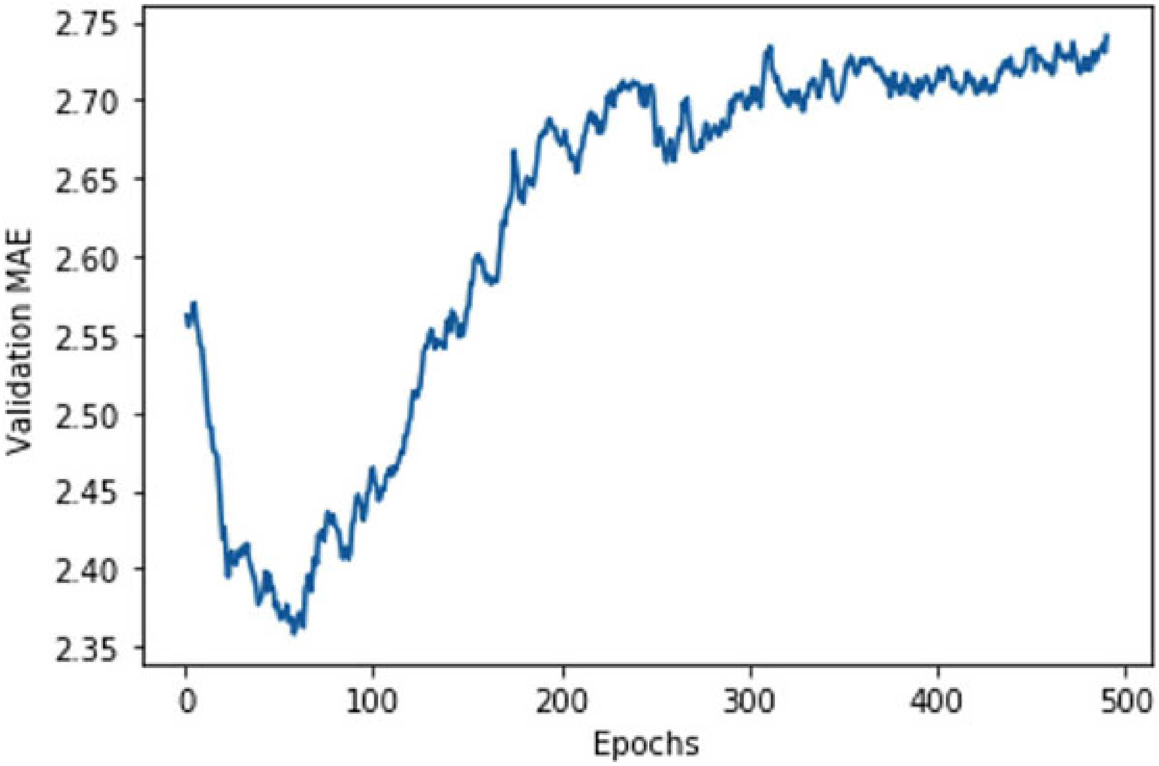


Figure 2.12: *Average MAE as a function of epochs after removing the first 10 items*

According to the *preceding plot*, validation MAE stops improving significantly after *80 epochs*. After that, it starts overfitting. Once we have tuned other parameters of our model (besides the number of epochs, we can also adjust the size of the hidden layers), we can train a *production* model on the training data, with the best parameters, and look at its performance on the **`test`** data:

```
model = build_model()
# Train it on the entirety of the data.
model.fit(train_data, train_targets,
    epochs=80, batch_size=16, verbose=0)
test_mse_score, test_mae_score = model.evaluate(test_data,
test_targets)
```

The **`test_mae_score`** value we got is **`2.6080945`**. Looking at the **`test_mae_score`**, it is still quite high.

Overfitting and underfitting

In the previous section, we saw that the accuracy of the model on the validation data peaks after training for a number of epochs, and would then stagnate or start decreasing. This meant, our model would overfit on the training data. It is important to know how to deal with overfitting. It is often possible to achieve high accuracy on the training set, what we want is to develop models that generalize to a testing data set (data model hasn't been seen before). The opposite of overfitting is called **underfitting**. It occurs when there is room for improvement on the test data. It can happen for reasons like: if the model is not powerful enough, is over-regularized, or has simply not been trained long enough. It means the network has not learned the relevant patterns in the training data. If you train for too long, the model starts to overfit, and will learn patterns from the training data that might not generalize to the test data. We will strike a balance. Being able to understand how to train for an appropriate number of epochs, as we'll explore in the following section, is an important skill. The best solution to prevent overfitting is to use a more complete training dataset. The training dataset should cover the full range of inputs that the model is expected to handle. Getting additional data is useful if it covers new cases.

A model that is trained on more complete data will generalize better. The next best solution available is to use techniques like regularization. This places constraints on the information (quantity and type) your model can store. When a network can only memorize a small number of patterns, the optimization process will force it to focus on the most prominent and generalized patterns.

In this section, we'll explore several common regularization techniques, and use them to improve on a classification model.

Sonar dataset

This dataset consists of *208 rows* and *60 features* for sonar readings to classify an object to be a **Rock** (**R**) or **Metal** (**M**). The features are a number between *0.0* and *1.0* and represent energy within a particular frequency band.

Reference:

Let us first load the dataset using **`pandas`**:

```
dataframe = read_csv("../data/sonar.all-data", header=None)
dataset = dataframe.values
dataset
```

The dataset will contain all the values loaded:

```
array([[0.02, 0.0371, 0.0428, ..., 0.009, 0.0032, 'R'],
   [0.0453, 0.0523, 0.0843, ..., 0.0052, 0.0044, 'R'],
   [0.0262, 0.0582, 0.1099, ..., 0.0095, 0.0078, 'R'],
   ...,
   [0.0522, 0.0437, 0.018, ..., 0.0077, 0.0031, 'M'],
   [0.0303, 0.0353, 0.049, ..., 0.0036, 0.0048, 'M'],
   [0.026, 0.0363, 0.0136, ..., 0.0061, 0.0115, 'M']],
   dtype=object)
```

Let us look at the shape of the dataset using **`dataset.shape`**. It gives an output of (*208, 61*) which is what we were expecting. The last column is the label, which we can check using **`dataset[:60]`**, you will get a **`NumPy`** array with values **`array(['R', 'R', 'R', ....', 'M',', 'M', 'M'],dtype=object)`**.

Next, we will split the dataset into input (**`X`**) and output (**`Y`**) variables and encode the label **`Y`** using **`sklearn.preprocessing.LabelEncoder`**:

```
from sklearn.preprocessing import LabelEncoder
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
encoded_Y
```

`encoded_Y` `will look like` **`array([1, 1, 1, 1, ...0, 0])`**.

LabelEncoder (**https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder**) is a utility class which normalizes labels so that they contain only values between **0** and n_classes. LabelEncoder can be used as follows:

```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
le.fit([1, 2, 2, 3,3,3])
le.classes_
Output of the classes:
array([1, 2, 3])
le.transform([1, 1, 2, 3,3,3])
```

The output, after transforming the input, shows how it has been encoded into 0, 1, and 2:

```
array([0, 0, 1, 2, 2, 2]).
```

Let us split the `X` and `encoded_Y` into `train` and `test` dataset:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, encoded_Y,
test_size=0.25, random_state=0)
```

First, we will create a sequential model with no hidden layer. We have an input layer of *10 neurons*, with dimension of *60* (matching the number of features). It has an activation of ReLU applied to it. This layer feeds into a dense layer which is the output layer with *1 neuron* and activation function **Sigmoid**. We will use a basic **SGD optimizer** and use accuracy as the metrics.

SGD calculates the gradient descent using local gradient:

$$\theta_{t+1} = \theta_t - \alpha \nabla Q_\theta J\left(\theta; x^{(i)}, y^{(i)}\right)$$

Where $J^{(\theta)}$ is the objective function and θ is the input parameter to the cost function:

```
def create_tiny_model():
  # create model
   model = Sequential()
   model.add(Dense(10, input_dim=60, activation='relu'))
   model.add(Dense(1, activation='sigmoid'))
   # Compile model
   sgd = SGD(lr=0.01, momentum=0.8)
   model.compile(loss='binary_crossentropy', optimizer=sgd,
   metrics=['accuracy'])
   return model
```

Let us now train the model by calling model.fit and capture the history:

```
ckpt_tiny_model = 'sonar_tiny.best.hdf5'
tiny_model_checkpoint = ModelCheckpoint(ckpt_tiny_model,
        monitor='val_accuracy',
        verbose=1,
        save_best_only=True,
        mode='max')
callbacks_list = [tiny_model_checkpoint]
tiny_model = create_tiny_model()
print('Starting training:')
# train the model, store the results for plotting
tiny_model_history = tiny_model.fit(X_train,
        y_train,
        validation_data=(X_test, y_test),
        epochs=EPOCHS,
        batch_size=BATCH_SIZE,
        callbacks=callbacks_list,
        verbose=0)
```

Let us plot the history obtained from the preceding model. We will define a generic function **`plot_history(history)`** as listed:

```
%matplotlib inline
import matplotlib.pyplot as plt
def plot_history(history):
   print(history.history.keys())
   # summarize history for accuracy
   plt.plot(history.history['accuracy'])
   plt.plot(history.history['val_accuracy'])
   plt.title('model accuracy')
   plt.ylabel('accuracy')
   plt.xlabel('epoch')
   plt.legend(['train', 'test'], loc='upper left')
   plt.show()
   # summarize history for loss
   plt.plot(history.history['loss'])
   plt.plot(history.history['val_loss'])
```

```
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()
plot_history(tiny_model_history)
```

As can be seen from the following *figure 2.13*, the accuracy stops at around `0.86`:

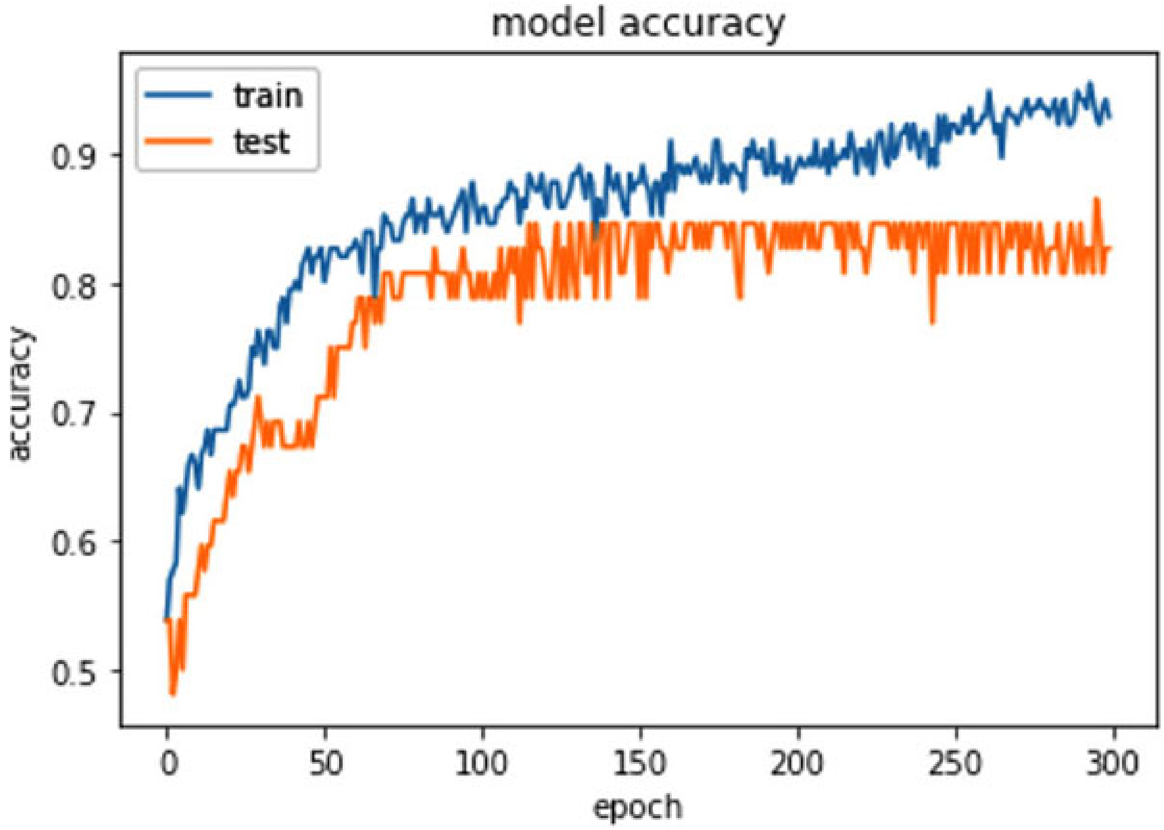


Figure 2.13: *Tiny model accuracy for training and test dataset*

Validation loss on the other hand bottoms out and again increases, which means the model is overfitting (refer to *figure 2.14*, which plots model loss for **train** and **test** data):

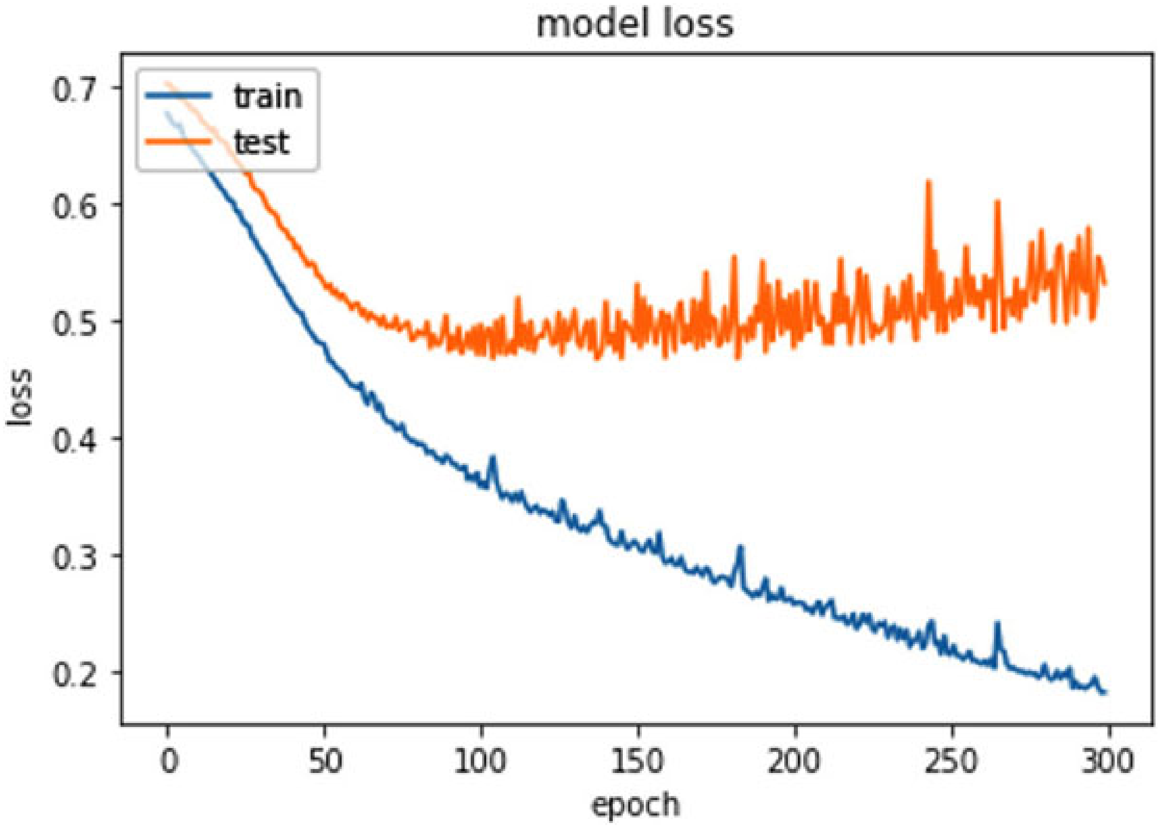


Figure 2.14: *Tiny model loss for training and test dataset*

Next, let us create a baseline model with one hidden layer with `30` neurons and ReLU activation:

```
def create_baseline():
  # create model
   model = Sequential()
   model.add(Dense(60, input_dim=60, activation='relu'))
   model.add(Dense(30, activation='relu'))
   model.add(Dense(1, activation='sigmoid'))
   # Compile model
   sgd = SGD(lr=0.01, momentum=0.8)
   model.compile(loss='binary_crossentropy', optimizer=sgd,
   metrics=['accuracy'])
   return model
```

Let us look at the model validation accuracy and loss for base line model. It is plotted in *figure 2.15*:

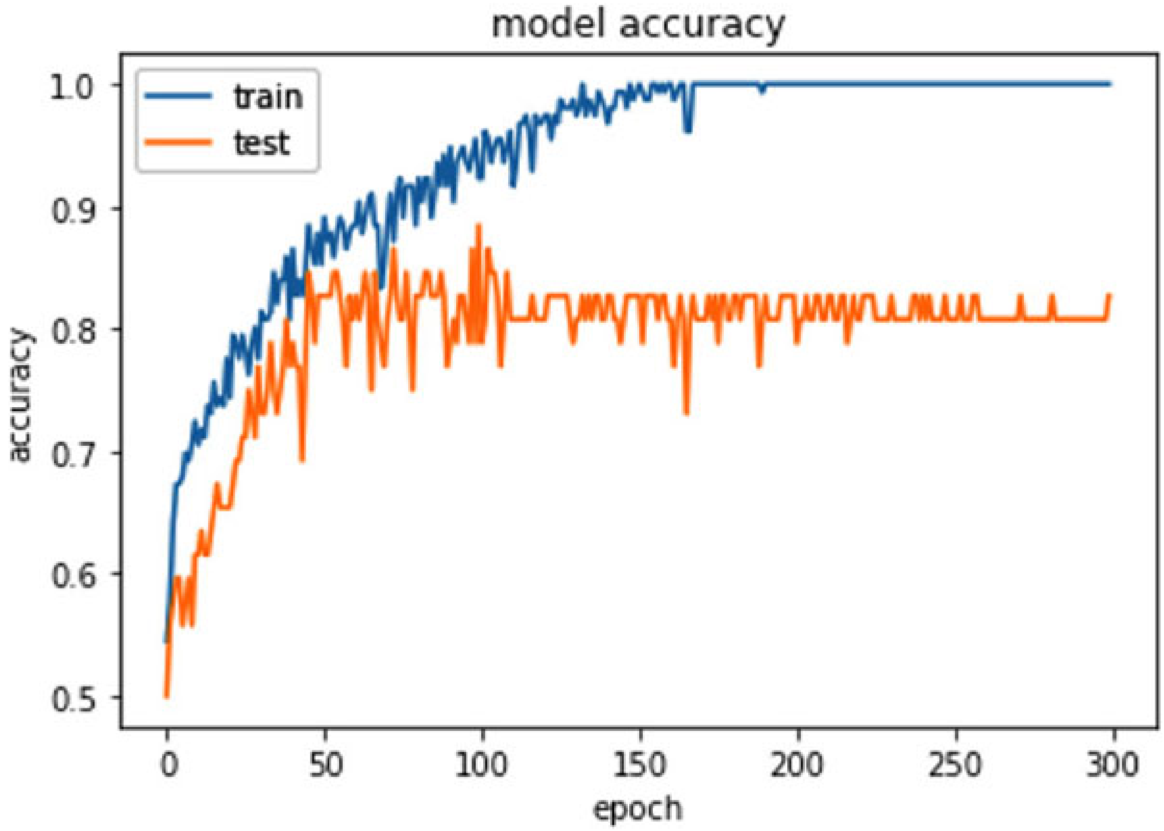


Figure 2.15: *Base model accuracy for training and test dataset*

Comparing this graph with the previous one, it is clear that while training accuracy tends to get much better validation, accuracy is almost the same. *Figure 2.16* shows the loss for the preceding model:

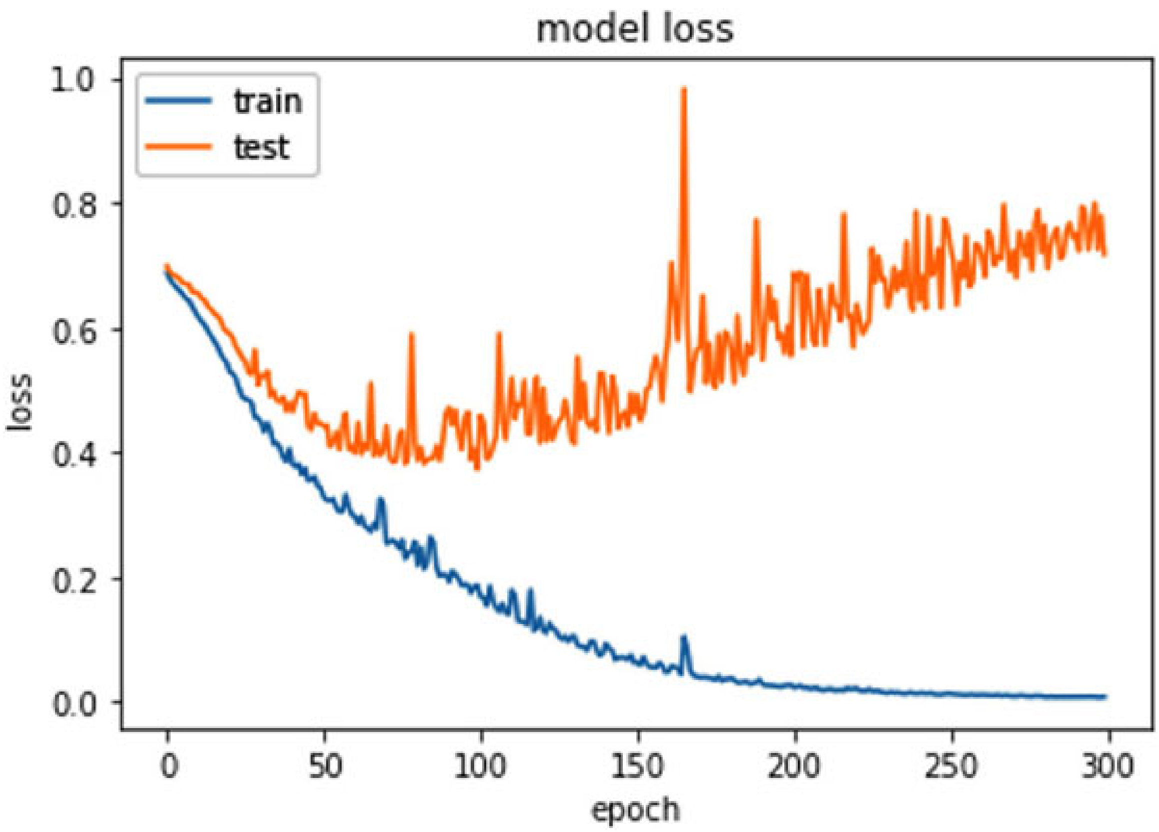


Figure 2.16: *Base model loss for training and test dataset*

Next, we are going to create a model with *L2 regularization*. The following note provides more details on L2 regularization.

Note: Regularization is an important technique in machine learning to prevent overfitting. Mathematically speaking, it adds a regularization term in order to prevent the coefficients from overfitting. The difference between the *L1* and *L2* is just that *L2* is the sum of the squares of the weights, while *L1* is just the sum of the weights.*L1* and *L2* regularization owe their names to the *L1* and *L2* norms of a vector, as shown in the following equation:

$$\left|\left|w\right|\right|_1 = \left|w_1\right| + \left|w_2\right| + \ldots + \left|w_n\right|$$

1-norm (also known as **L1 norm**)

$$\left|\left|w\right|\right|_2 = (w_1^2 + w_2^2 + \ldots + w_n^2)^{1/2}$$

1-2norm (also known as **L2 norm**)

If we define our predicted value using the following equation:

$$\hat{y} = wx + b$$

Loss function can be defined as: $L = (\hat{y} - y)^2$

Loss function with *L1* regularization can be defined as:

$$L = (\hat{y} - y)^2 + \lambda|w|$$

Then, loss function with *L2* regularization will be:

$$L = (\hat{y} - y)^2 + \lambda w^2$$

The value of is calculated manually, if it is greater than *0*.

The model with L2 regularization will be composed as shown by using **`tensorflow.keras.regularizers.l2`**:

```
from tensorFlow.keras import regularizers
def create_l2_model():
  # create model
  model = Sequential()
  model.add(Dense(60, input_dim=60, activation='relu',
      kernel_regularizer=regularizers.l2(0.001)))
  model.add(Dense(30, activation='relu',
      kernel_regularizer=regularizers.l2(0.001)))
  model.add(Dense(1, activation='sigmoid'))
  # Compile model sgd = SGD(lr=0.01, momentum=0.8)
  model.compile(loss='binary_crossentropy',
       optimizer=sgd,
       metrics=['accuracy'])
  return model
```

After *30 epochs,* the model gave the following validation accuracy:

```
Epoch 00300: val_accuracy did not improve from 0.86538
```

Next, we will implement dropout and L2 regularization together. Notice how **`tensorFlow.keras.layers.Dropout`** is being used as a new layer:

```
from tensorFlow.keras.layers import Dropout
def create_dropout_model():
  # create model
   model = Sequential()
```

```
    model.add(Dense(60, input_dim=60, activation='relu',
        kernel_regularizer=regularizers.l2(0.001)))
    model.add(Dropout(0.7))
    model.add(Dense(30, activation='relu',
        kernel_regularizer=regularizers.l2(0.001)))
    model.add(Dropout(0.8))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    sgd = SGD(lr=0.01, momentum=0.8)
    model.compile(loss='binary_crossentropy', optimizer=sgd,
    metrics=['accuracy'])
    return model
```

Dropout is a regularization method which approximates training a large number of neural networks with different architectures in parallel. During training, some of the outputs are randomly ignored or *dropped out*. This has an effect of making the layer look like and be treated like a layer with a different number of nodes and connections to the previous layer. Each update to a layer during training is performed with a different *view* of the configured layer.

Let us look at the validation accuracy for the model with drop out, refer to *figure 2.17*:

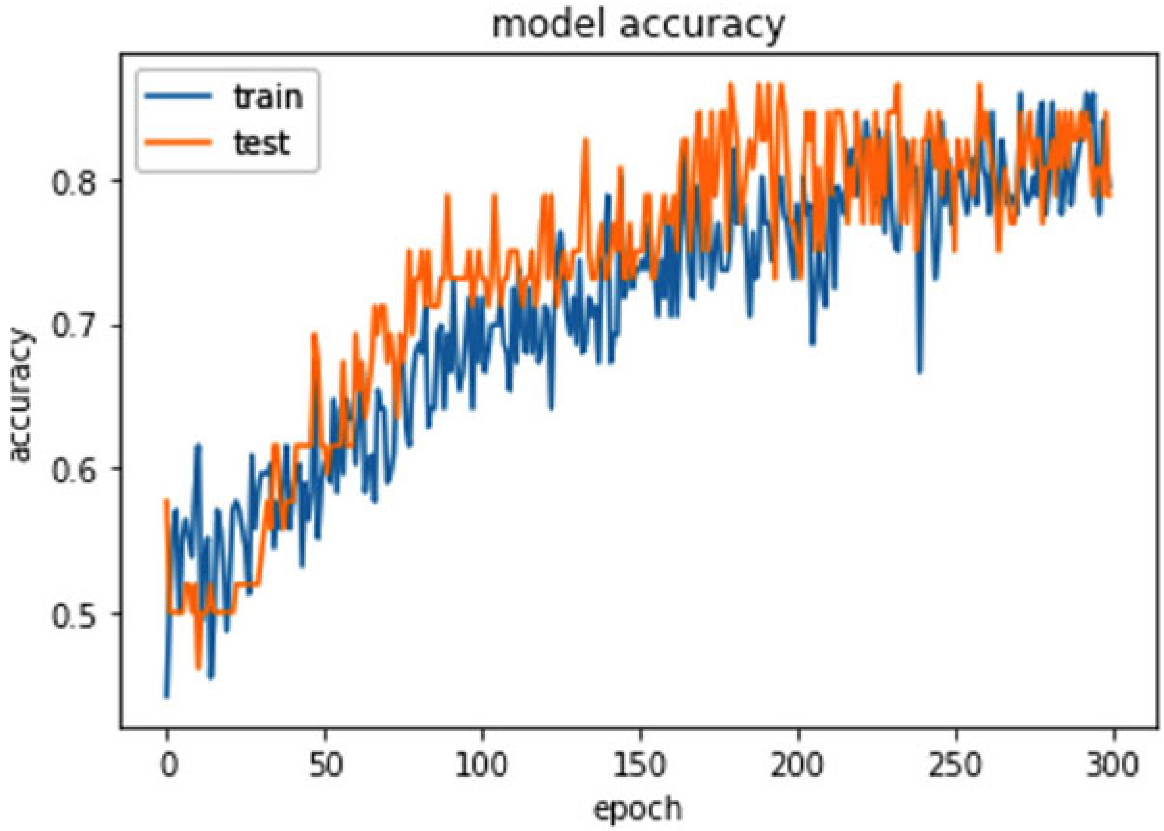


Figure 2.17: *Dropout model accuracy for training and test dataset*

Interestingly, the final number matches the preceding approaches after *300 epochs*:

```
Epoch 00300: val_accuracy did not improve from 0.86538
```

Let us look at the loss for this model configuration, which is plotted in *figure 2.18*:

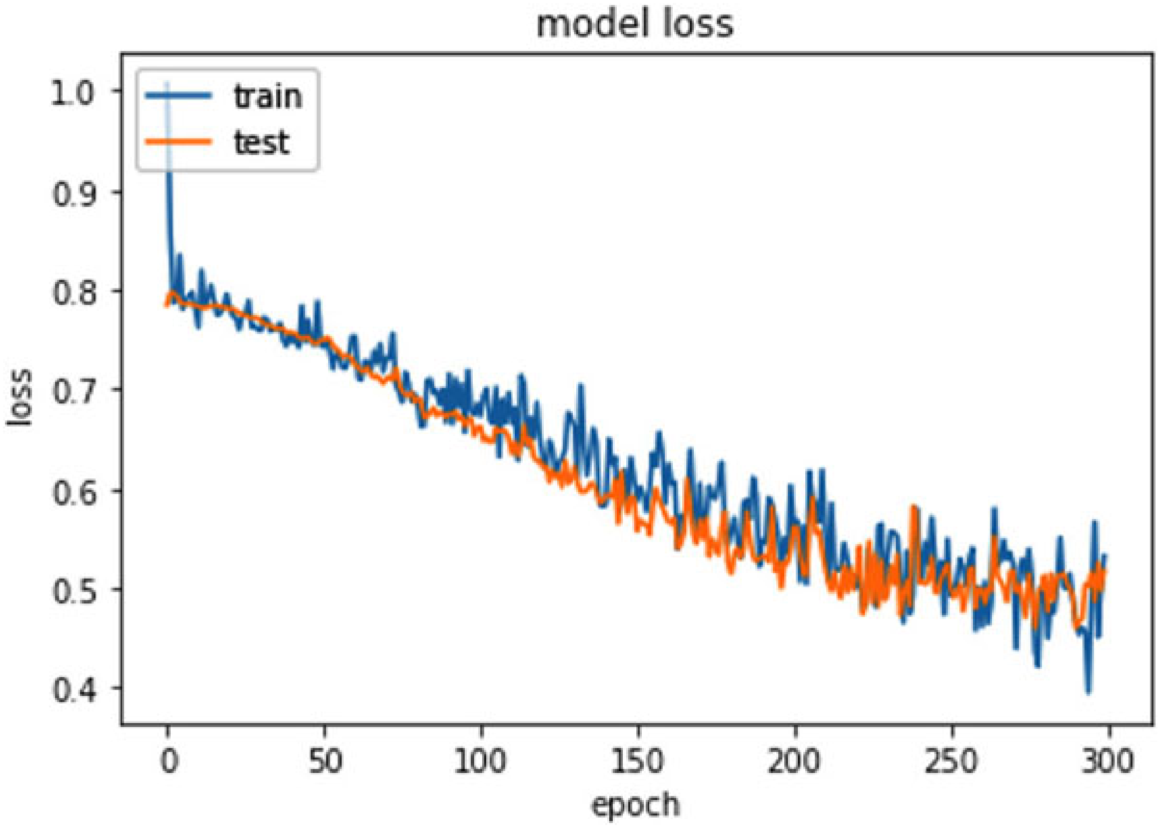


Figure 2.18: *Dropout model loss for training and test dataset*

As can be seen from the preceding diagram, while the validation accuracy is similar the validation loss is much closer to the training loss if we apply dropout to the model.

Saving and restoring models

Let us look at how to save and restore models:

```
from __future__ import absolute_import, division, print_function,
unicode_literals
import TensorFlow as tf
tf.keras.backend.clear_session()
```

1. First, let us create a basic model for processing **MNIST** dataset:

   ```
   from tensorFlow import keras
   from tensorFlow.keras import layers
   inputs = keras.Input(shape=(784,), name='digits')
   x = layers.Dense(64, activation='relu', name='dense_1')
   (inputs)
   x = layers.Dense(64, activation='relu', name='dense_2')(x)
   ```

```
outputs = layers.Dense(10, activation='softmax',
name='predictions')(x)
model = keras.Model(inputs=inputs, outputs=outputs,
name='3_layer_mlp')
model.summary()
```

2. Let us compile the model using **`model.compile(..)`** and then train the model using **`model.fit(..)`**:

   ```
   (x_train, y_train), (x_test, y_test) =
   keras.datasets.mnist.load_data()
   x_train = x_train.reshape(60000, 784).astype('float32') /
   255
   x_test = x_test.reshape(10000, 784).astype('float32') / 255
   model.compile(loss='sparse_categorical_crossentropy',
       optimizer=keras.optimizers.RMSprop())
   history = model.fit(x_train, y_train,
          batch_size=64,
           epochs=1)
   ```

3. Next, we will call **`model.predict(..)`** to get predictions. We will use these predictions to compare with the saved and loaded model:

   ```
   predictions = model.predict(x_test)
   ```

4. Finally, let us save this model:

   ```
   model.save('sample_model.h5')
   ```

Let us look at the API in more detail:

```
save(
   filepath,
   overwrite=True,
   include_optimizer=True,
   save_format=None,
   signatures=None,
   options=None
)
```

This function saves the model to TensorFlow, saved as a model or a single **HDF5** file.

The saved file includes:

- the model architecture, allowing to re-instantiate the model.
- the model weights.

- the state of the optimizer, allowing it to resume training exactly where it was left off.

Let us look at the parameters in more detail:

- **`filepath`**: String, path to saved model or H5 file to save the model.
- **`overwrite`**: Whether to silently overwrite any existing file at the target location, or provide the user with a manual prompt.
- **`include_optimizer`**: If **`True`**, save optimizer's state together.
- **`save_format`**: Either **`tf`** or **`h5`**, indicating whether to save the model to TensorFlow saved model or HDF5.
- The default for TensorFlow 1.0 was **`h5`**, but will switch to **`tf`** in TensorFlow 2.0/2.1. Now, let us try to recreate the model from the preceding file:

  ```
  # Recreate the exact same model purely from the file
  new_model = keras.models.load_model('sample_model.h5')
  ```

Let us check the state and compare with older predictions:

```
import numpy as np
# Check that the state is preserved
new_predictions = new_model.predict(x_test)
np.testing.assert_allclose(predictions, new_predictions, rtol=1e-
6, atol=1e-6)
```

Note that the optimizer state is preserved as well: *you can resume training where you left off*.

Let us also try to compare visually:

```
prediction[:1] array([[2.0078852e-04, 3.8760394e-08, 2.0996991e-
03, 2.3580731e-03, 1.7406029e-07, 4.1207833e-05, 4.5558304e-09,
9.9482697e-01, 2.5986974e-05, 4.4698527e-04]], dtype=float32)
new_predictions[:1] array([[2.0078852e-04, 3.8760394e-08,
2.0996991e-03, 2.3580731e-03, 1.7406029e-07, 4.1207833e-05,
4.5558304e-09, 9.9482697e-01, 2.5986974e-05, 4.4698527e-04]],
dtype=float32)
```

As you can see, the values match. Hence, the model has been persisted and recreated successfully.

Similarly, there are more advanced APIs for saving the weights or the configuration alone.

If you are only interested in the architecture of the model, and don't need to save the weight values or the optimizer, you can retrieve the **`config`** of the model via the **`get_config()`** method. The **`config`** is a Python dictionary (**`dict`**) that enables you to recreate the same model: initialized, without any of the information/weights learned previously during training:

```
# save and reload architecture
config = model.get_config()
reinitialized_model = keras.Model.from_config(config)
```

With this, we come towards the end of the chapter.

Conclusion

We learnt how to use TensorFlow 2.x for classification and regression, using basic structured text examples. We also learnt how to deal with underfitting and overfitting by using techniques like L2 regularization and dropout. We also learnt persisting and re-loading persisted models, and various formats supported.

In the next chapter, we will learn more about the Keras functional APIs, which have been inducted into TensorFlow 2.x. We will also dive deep into optimizers and losses supported.

Question

Use TensorFlow 2.x to do sentiment detection using supervised learning, please refer to dataset at the following link:

https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences

and some examples and articles on older Keras based APIs in the following link:

https://realpython.com/python-keras-text-classification/

CHAPTER 3
Keras Based

Introduction

In this chapter, we are going to do a deep dive on Keras' functional APIs, as implemented on TensorFlow 2.x. Keras' functional API's ease of use is one of the main reasons why Keras became successful. We look at the convergence of two libraries in this chapter, which aims to be a foundation for the rest of the book. We will start the chapter with training evaluation and prediction APIs. We will then look at loss, metrics, and optimizer implementations in this higher-level API. In the end, we will look at some of the changes made with 2.x like namespaces.

Structure

In this chapter, we will cover the following topics:

- Keras functional API overview
    - The relation between Keras and TensorFlow
    - *How Keras initializes the TensorFlow graph?*
    - *How the session is linked to the Keras backend?*
    - Sequential models and their parents (**`tensorflow.python.keras.engine.training.model`**)
    - Keras layers: base layer
- Training evaluation and prediction using Keras APIs
- Loss functions
- Optimizers
- Training, evaluation, and prediction using TensorFlow and Keras
    - *Here we will cover using built-in evaluation loops*
    - **`model.compile`** and **`model.fit`**

- Layers and models using Keras functional APIs
- Deep dive into optimizers, loss functions, and metrics supported by Keras functional APIs
- Effective TensorFlow 2.x

Objective

The objective of this chapter is to make the readers familiar with the key Keras functional API concepts like **layers** and how training evaluation and prediction are exposed, and used. We will also look at loss, metrics, and optimizers supported by TensorFlow 2.x. In the end, we will look at effectively using TensorFlow 2.x APIs, and how some of the concepts like global variables, sessions, and so on have been done away with.

Keras functional API

Keras' sequential API is a popular mechanism for creating deep learning models. However, the programming model comes with the assumption that it takes one input and will have only one output. Models with this type of structure are of no use when more than one input and output is required. Some deep learning networks might have multiple internal branches (for example, **inception module**), which is quite different from the traditional sequential model. To deal with such architectures, the functional API of Keras is quite handy. This functional API paradigm is natively supported by TensorFlow 2.x.

Training, evaluation, and prediction using TensorFlow and Keras

This section covers training, evaluation, and prediction (inference) models in TensorFlow 2.x in two broad situations:

- when using built-in APIs for training and validation (such as **`model.fit()`**, **`model.evaluate()`**, **`model.predict()`**)
- when writing custom loops from scratch using eager execution and the **`GradientTape`** object

Whether you are using built-in loops or writing your own, model training, and evaluation works in the same way across every kind of Keras model -

sequential models, **models built with the functional API**, and **models written from scratch via model subclassing**.

Using training and evaluation loops

While passing data to the built-in training loops of a TensorFlow 2.x model, you use NumPy arrays (if your data is small and can fit in memory) or **`tf.data`** dataset objects. In the next few sections, we'll use the **Fashion MNIST** dataset as NumPy arrays, to demonstrate how to use optimizers, losses, and metrics:

1. We will be using **`wandb`** to collect and display the metrics as well as the TensorBoard:

   ```
   from tensorflow import keras
   from tensorflow.keras import layers
   import wandb
   from wandb import magic
   wandb.init(project="tf_book_ch3_training_eval_fashion_mnist"
   )
   inputs = keras.Input(shape=(784,), name='digits')
   x = layers.Dense(64, activation='relu', name='dense_1')
   (inputs)
   x = layers.Dense(64, activation='relu', name='dense_2')(x)
   outputs = layers.Dense(10, name='predictions')(x)
   model = keras.Model(inputs=inputs, outputs=outputs)
   ```

   Here is what the typical end-to-end workflow looks like, consisting of training, validation on a training sub-set generated from the original training data, and evaluation on the test data.

2. Load a sample dataset for the sake of this example:

   ```
   (x_train, y_train), (x_test, y_test) =
   keras.datasets.fashion_mnist.load_data()
   # Preprocess the data (these are Numpy arrays)
   x_train = x_train.reshape(60000, 784).astype('float32') /
   255
   x_test = x_test.reshape(10000, 784).astype('float32') / 255
   y_train = y_train.astype('float32')
   y_test = y_test.astype('float32')
   # Reserve 10,000 samples for validation
   x_val = x_train[-10000:]
   y_val = y_train[-10000:]
   ```

```
x_train = x_train[:-10000]
y_train = y_train[:-10000]
```

3. Specify the training configuration (**`optimizer`**, **`loss`**, **`metrics`**):

```
model.compile(optimizer=keras.optimizers.RMSprop(), #
Optimizer
      # Loss function to minimize
      loss=keras.losses.SparseCategoricalCrossentropy(
      from_logits=True),
      # List of metrics to monitor
      metrics=['sparse_categorical_accuracy'])
```

4. Slice the data into *batches* of size **`batch_size`**, and iterate over the entire dataset for a specified number of **`epochs`** to train the model:

```
print('# Fit model on training data')
history = model.fit(x_train, y_train,
       batch_size=64,
       epochs=3,
       # We pass some validation for
       # monitoring validation loss and metrics
       # at the end of each epoch
       validation_data=(x_val, y_val))
print('\nhistory dict:', history.history)
history dict: {'loss': [0.5624035213088989,
0.3971926462173462, 0.3580621465587616],
'sparse_categorical_accuracy': [0.8013, 0.85456, 0.87052],
'val_loss': [0.5479889378070831, 0.43700868356227873,
0.4053771690368652], 'val_sparse_categorical_accuracy':
[0.7936, 0.843, 0.8553]}
```

We can see the graph in the TensorBoard at the following location:

The following *figure 3.1* shows inputs specified by **digits**, **digits_1**, **digits_2**, and the associated TensorFlow graph:

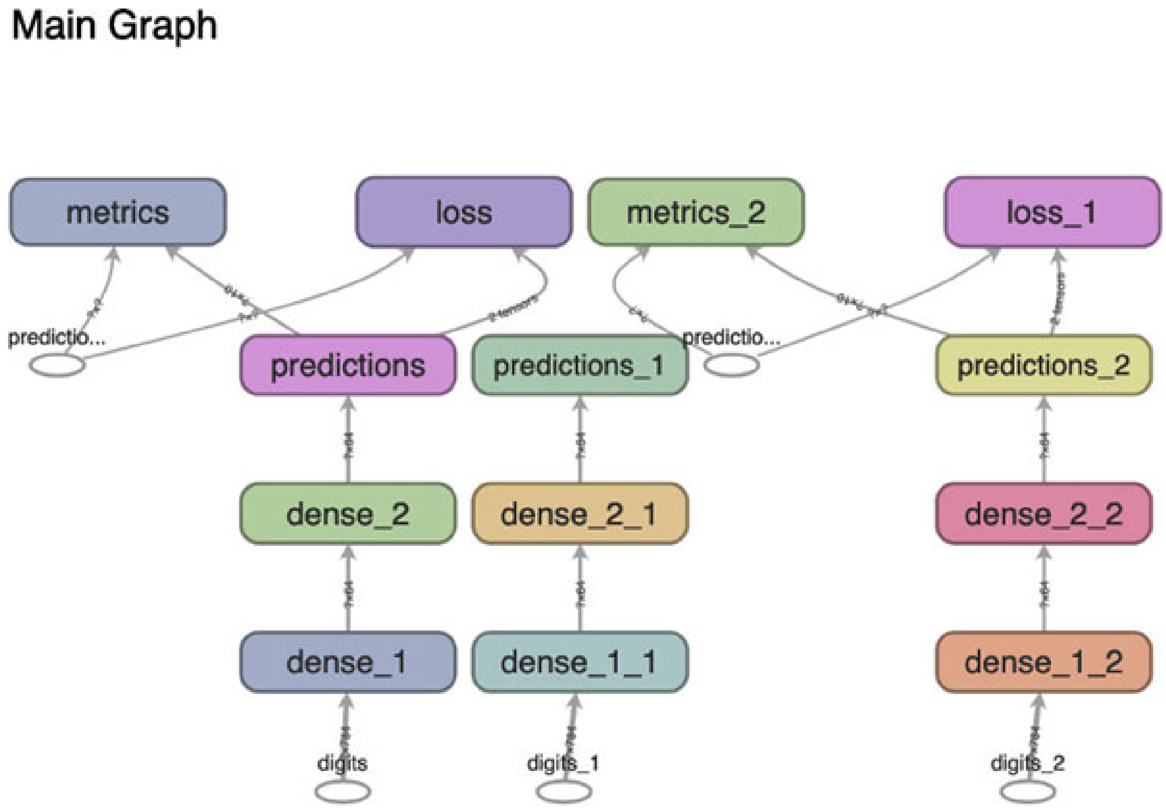
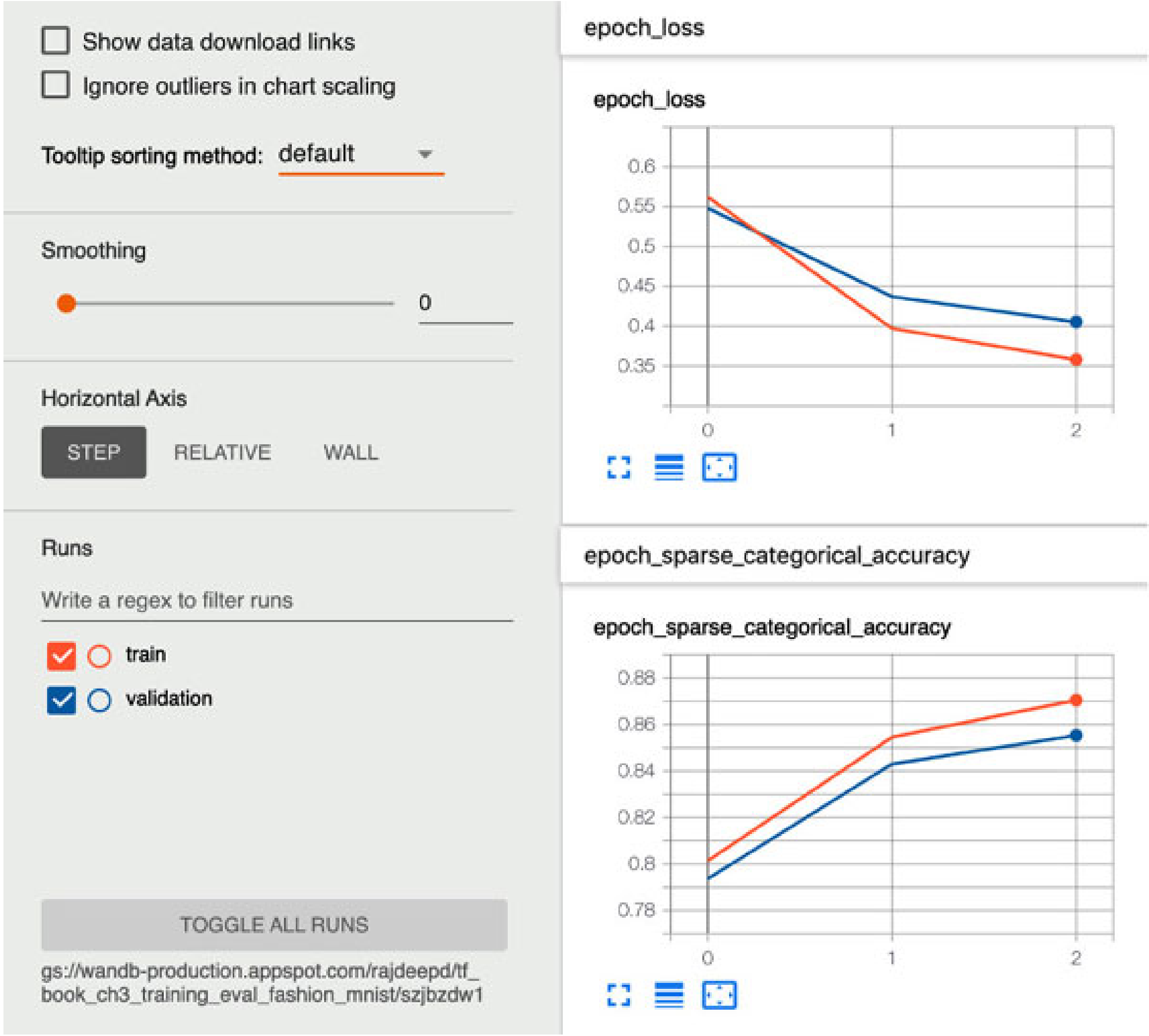


Figure 3.1: *TensorBoard showing the TensorFlow graph with two hidden layers and three runs specified by inputs with digits, digits_1, and digits_2*

Let us look at the training and evaluation graphs as a function of epochs as shown in *figure 3.2*:



□ Show data download links

□ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0

Horizontal Axis: STEP

Runs

Write a regex to filter runs

☑ ○ train

☑ ○ validation

gs://wandb-production.appspot.com/rajdeepd/tf_book_ch3_training_eval_fashion_mnist/szjbzdw1

Figure 3.2: *TensorBoard scalar graph showing the training and validation loss and accuracy*

(You can find more details at the link specified previously.)

Loss, metrics, and an optimizer

To train a TensorFlow model with fit, you need to specify a **`loss`** function, an **`optimizer`**, and one or more **`metrics`** to monitor.

These parameters are passed to the model as arguments to the **`compile()`** method:

```
model.compile(optimizer=…, # Optimizer
        # Loss function to minimize
        loss=…
        # List of metrics to monitor
        Metrics=..)
```

The metrics argument is a list—the model can have any number of metrics.

If the model has multiple outputs, different losses and metrics can be specified for each output. You can also modulate the contribution of each output to the total loss of the model.

- **Loss**: For optimization algorithms, the function used to evaluate a set of weights is called the **objective function**.

  We may want to maximize or minimize the objective function, that we are searching for a set of weights that gives the highest or lowest score respectively.

  With neural networks, we want to minimize the error. The objective function is referred to as a **cost function** or a **loss function**. The value which is calculated by the loss function is called simply a **loss**.

  The function to be minimized or maximized is called the objective function. When we are minimizing it, we also call it the cost function, loss function, or error function.

  — Page 82,

  Deep Learning (Adaptive Computation and Machine Learning series) Illustrated Edition, 2016

  by Ian Goodfellow , Yoshua Bengio, Aaron Courville

  The cost or loss function must distill all aspects of the model down into a single number in a way that improvements in that value is a sign of a better model.

  The cost function reduces all the aspects of a complex system down to a single number: a scalar value, which allows models (also referred to as candidate solutions) to be ranked and compared.

  — Page 155,

  Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks, 1999.

  To calculate the error of the model during the optimization process, a loss function is chosen.

  This is a challenging problem as the function must capture the properties of the problem and take stock of the concerns important to the project and stakeholders.

> *The function has to represent our design goals in a faithful manner. If we have chosen a poor error function and get unsatisfactory results, the fault is ours.*
>
> *— Page 155,*
>
> *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks, 1999.*

Given that we are familiar with the loss function and loss, we need to know what functions to use. This is the same problem we are trying to solve with the neural network.

Given a training dataset with one or more input variables, we want a model to estimate weight parameters that best map examples of the inputs to the output or target variable.

Given an input dataset, the model tries to make predictions that match the data distribution of the target variable. Using maximum likelihood, a loss function estimates how closely the distribution of predictions made by a model matches the actual distribution of target variables in the training data.

The benefit of using maximum likelihood as a framework to estimate the model parameters (weights), for neural networks and in machine learning, is that as the number of training examples is increased, the estimate of the model parameters also improves. This property is called the **property of consistency**.

- **Maximum likelihood and cross-entropy**: With the framework of maximum likelihood, the error between two probability distributions is measured using cross-entropy.

  For a classification problem, we are interested in mapping input variables to a class label. The problem is to predict the probability of an example belonging to each class. In a *binary classification*, there would be two classes, so we can predict the probability of the example belonging to the first class. In case of *multiple-class classification*, we can predict probability for the example data belonging to each of the classes.

  The probability of an example belonging to a given class is **1** or **0** in the training dataset, as each sample is a known example, and labelled.

  Therefore, under maximum likelihood estimation, we want to determine model weights that minimize the difference between the model's predicted probability distribution for the given dataset and the distribution of probabilities in the training dataset. This is called the **cross-entropy**.

Cross-entropy comes from information theory and has the unit of *bits*. It has been used to estimate the difference between estimated and predicted probability distributions.

In the case of regression problems, a number is predicted. Hence, it is common to use **mean squared error** (**MSE**) loss function instead of cross-entropy.

> *The MSE is populist for function approximation (regression) problems. The cross-entropy error function is selected for classification problems where predictions are defined as probabilities of data belonging to a specific class.*
>
> *— Page 155-156,*
>
> *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks, 1999.*

Under maximum likelihood estimation and assuming a Gaussian distribution for the target variable, MSE can be assumed to be the cross-entropy between the distribution of the model predictions and the distribution of the actual target variable.

> *Any loss derived from negative log-likelihood is a cross-entropy between the empirical distribution of the training set and the probability distribution predicted by model. As an example, MSE is the cross-entropy between the empirical distribution and a Gaussian model.*
>
> *— Page 132, Deep Learning (Adaptive Computation and Machine Learning series) Illustrated Edition, 2016*
>
> *by Ian Goodfellow , Yoshua Bengio, Aaron Courville*

When using maximum likelihood estimation, we will implement a cross-entropy loss function, which means we will use a cross-entropy loss function for classification problems and a MSE loss function for regression problems.

Deep learning neural networks are usually trained under the framework of maximum likelihood using cross-entropy as the loss function.

- **Which loss function to use**: We can summarize the previous section, and suggest the loss functions that should be used under a framework of maximum likelihood.

> The choice of the loss function is related to the activation function used in the output layer of the neural network. These two elements are connected.
>
> Configuration of the output layer is a choice of framing the prediction problem, the choice of the loss function as a means to calculate the error for a given framed problem.
>
> *The choice of a cost function is closely coupled with the choice of output unit. We generally use the cross-entropy between the training data distribution and the model distribution as a determinant. The choice of output representation determines the form of the cross-entropy function.*
>
> *— Page 181, Deep Learning (Adaptive Computation and Machine Learning series) Illustrated Edition, 2016*
>
> *by Ian Goodfellow , Yoshua Bengio, Aaron Courville*
>
> We will review the best practice/default values for each problem type with regard to the output layer and loss function, and also look at the corresponding implementations in TensorFlow 2.x.

Regression problem

A use case where we predict a real value:

- **Output layer**: one node with a linear activation unit.
- **Loss function**: **Mean Squared Error** (**MSE**).

Binary classification problem

A use case where we classify an example as belonging to one of two classes.

The problem is to predict the likelihood of an example belonging to class one, for example, the class that is assigned the integer value **1**, while the other class is assigned the value **0**:

- **Output layer**: one node with a sigmoid activation unit.
- **Loss function**: cross-entropy, also referred to as **logarithmic loss**.

Multi-class classification problem

A problem where we have to classify an example to belong to one of the classes.

This problem is framed as predicting the likelihood of an example belonging to each class:

- **Output layer configuration**: one node for each class using the **Softmax activation function**.
- **Loss function**: cross-entropy, also referred to as logarithmic loss.

Implementation of loss functions in TensorFlow 2.x

Let us look at implementation of supported loss functions in *TensorFlow 2.x* with simple examples.

Mean Square Error (MSE)

In statistics, the MSE or **mean squared deviation** (**MSD**) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value. MSE is a risk function, corresponding to the expected value of the squared error loss. The fact that MSE is almost always strictly positive (and not zero) is because of randomness or because the estimator does not account for information that could produce a more accurate estimate.

The MSE is a measure of the quality of an estimator—it is always non-negative, and values closer to zero are better.

The MSE is the second moment of the error, and incorporates both the variance of the estimator (how widely spread the estimates are from one data sample to another) and its bias (how far off the average estimated value is from the truth). For an unbiased estimator, the MSE is the variance of the estimator. Like the variance, MSE has the same units of measurement as the square of the quantity being estimated.

MSE in TensorFlow is implemented using **`tf.keras.losses.MeanSquaredError`**:

```
mse_obj = losses.MeanSquaredError()
true = [1, 9, 2, -5, -2, 6]
pred = [4, 8, 12, 8, 1, 3]
y_true = constant_op.constant(true, shape=(2, 3))
y_pred = constant_op.constant(pred, shape=(2, 3),
  dtype=dtypes.float32)
loss = mse_obj(y_true, y_pred)
```

```
print(loss)
```

Output will be **`tf.Tensor(49.5, shape=(), dtype=float32).`**

Sparse cross categorical entropy

Use this cross-entropy loss function when there are two or more label classes. Labels should be provided as integers. For *one-hot representation*, categorical cross-entropy loss should be used. There should be **`#`** classes floating point values per feature for **`y_pred`** and a single floating-point value per feature for **`y_true`**.

In the following code snippet, there is a single floating-point value per example for **`y_true`** and **`#`** classes floating pointing values per example for **`y_pred`**. The shape of **`y_true`** is **`[batch_size]`** and the shape of **`y_pred`** is **`[batch_size, num_classes]`**:

```
t = backend.placeholder()
p = backend.placeholder()
cce = losses.sparse_categorical_crossentropy(t, p)
t_val = ops.convert_to_tensor_v2([0, 1, 2, 3])
p_val = ops.convert_to_tensor_v2([
  [.9, .05, .025,0.025],[.05, .89, .03,0.03],
  [.04, .01, .94,0.01],
  [.02, .02, .01,0.95]])
f = backend.function([t, p], cce)
result = f([t_val, p_val])
```

Let us look at cross categorical entropy next, where labels are provided as one-host representation.

Cross categorical entropy

You should use this cross-entropy loss function when there are two or more label classes. Labels need to be provided in a one-hot representation. There should be **`#`** classes floating point values per feature.

In the following snippet, there are **`#`** classes floating pointing values per example. The shape of both **`y_pred`** and **`y_true`** are **`[batch_size, num_classes]`**:

```
import tensorflow as tf
cce = tf.keras.losses.CategoricalCrossentropy()
loss = cce(
  [[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]],
  [[.9, .05, .05], [.05, .89, .06], [.05, .01, .94]])
print('Loss: ', loss.numpy())
```

Output will print the categorical cross-entropy value:

```
Loss: 0.09458993
```

Optimizers

Optimization algorithms help to minimize or maximize an objective function, which is a mathematical function dependent on models' parameters (which are learnable), and are used to compute **target values (y)** from **predictors (x)** used in the model.

As an example, we call weights and biases in a neural network as its internal learnable parameters.

Optimization algorithm falls in two major categories:

1. **First order optimization algorithms**: These algorithms minimize or maximize the loss function using the gradient value based on the parameters. Gradient descent is the most popular first order derivative that tells us whether the function is decreasing or increasing at a particular point. It gives us a tangential line to the point on error surface.
2. **Gradient function**: Gradient is a vector representing derivate ($(\frac{\partial x}{\partial y})$) that is instantaneous rate of change of *y* with respect to *x*. Gradient replaces a derivate if more than one variable is involved. It is calculated using **partial derivatives**. Gradient produces a vector field and is represented as a *Jacobian Matrix*.

Gradient descent

Gradient descent is the most popular optimization technique used for training machine learning and deep learning models. It tries to find the minima, controls the variance, and updates the model's parameters that lead to convergence.

$(\theta\,)J\,\Delta^{u}-\theta=\theta$ is the formula used to update the parameter where is learning rate.

$\nabla J(\theta)$ is the gradient of loss function - - $J(\theta)$ for parameters θ

Neural networks train using *back propagation* where we propagate forward *by calculating dot product of input signals and their weights,* and then apply activation function to those sums of products. After the calculation, we propagate backwards in the network, carry error terms, and update weights using gradient descent -we calculate gradient of **error** (**E**) with respect to **weights** (**W**) or the parameters and update the parameters(weights) in the opposite direction of gradient of the loss function. Figure 3.3 shows the shape of gradient – a plot between error **E** and weights **W**:

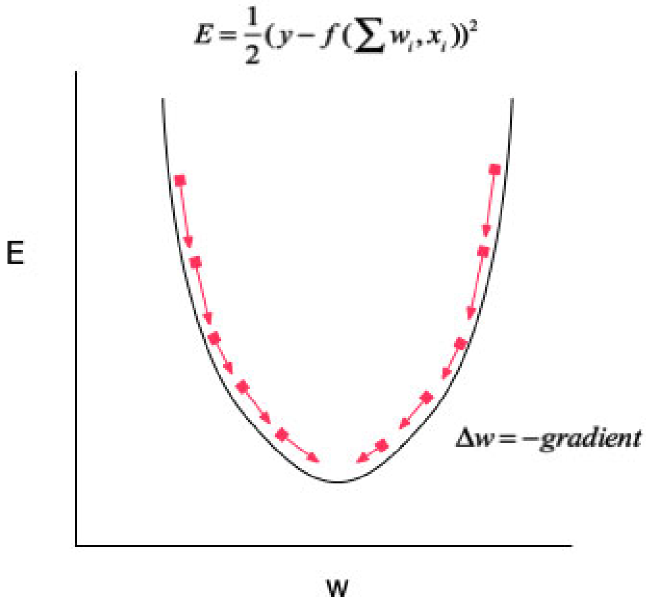


Figure 3.3: *The image shows process of updating the weights in opposite direction of gradient of error* $J(\theta)$.

U-shaped curve is the gradient. We update and optimize weights so that we reach local minima.

- **Batch gradient descent:** It calculates gradient of the whole dataset but performs one update. It is slow and hard to calculate for datasets that are large. Learning rate (*n*) determines how much update is required. This problem in standard gradient descent is rectified by using stochastic gradient descent.
- **Stochastic gradient descent (SGD):** SGD performs parameter updates for each training example, making it a much faster technique, even though the variations are much higher: $\theta=\theta-\eta\nabla J(\theta,x(i),y(i))$where $\{x(i),y(i)\}$are training parameters.

SGD applied to Fashion MNIST dataset using *TensorFlow 2.x* based APIs:

```
model_sgd = get_model()
model_sgd.compile(optimizer='sgd',
 loss='sparse_categorical_crossentropy',
 metrics=['accuracy'])
EPOCHS = 5
history_sgd = model_sgd.fit(X_train, y_train, epochs=EPOCHS,
 validation_data=(X_test, y_test))
```

If we plot the loss and validation loss, you can see the variations. Refer to the following *figure 3.4* where we show the training and validation loss for *20 epocs*:

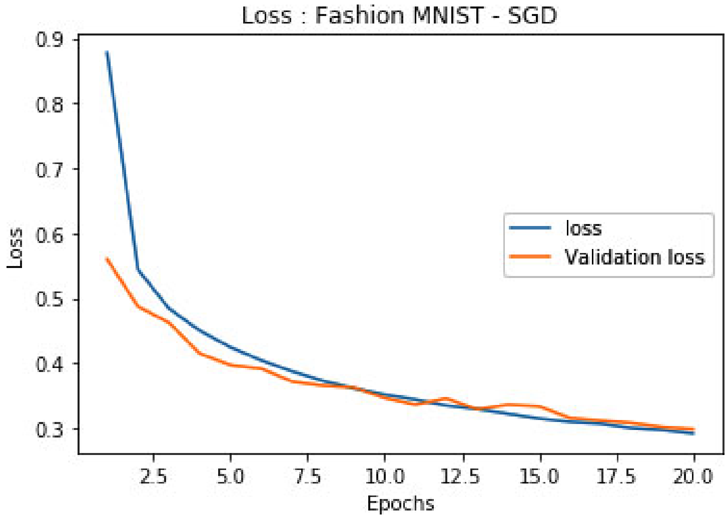


Figure 3.4: *Loss on training and validation dataset for SGD based optimizer on Fashion MNIST*

Next, let us look at techniques that improve upon basic SGD based approach:

- **Momentum based gradient descent**: Momentum based technique was invented to accelerate SGD by navigation along relevant direction and softening oscillations. This technique adds a fraction γ of the update vector of the last step to the current update vector: $V(t)=\gamma V(t-1)+\eta\nabla J^{(\theta)}$. And we update parameters to $\theta=\theta-V(t)$. Momentum is usually set to `0.9`.

  Let us see how momentum-based SGD compares to the normal SGD for Fashion MNIST dataset.

  The following *figure 3.5* shows that clearly the momentum-based SGD has lower training loss as compared to normal SGD, and also the training time is lower and faster:

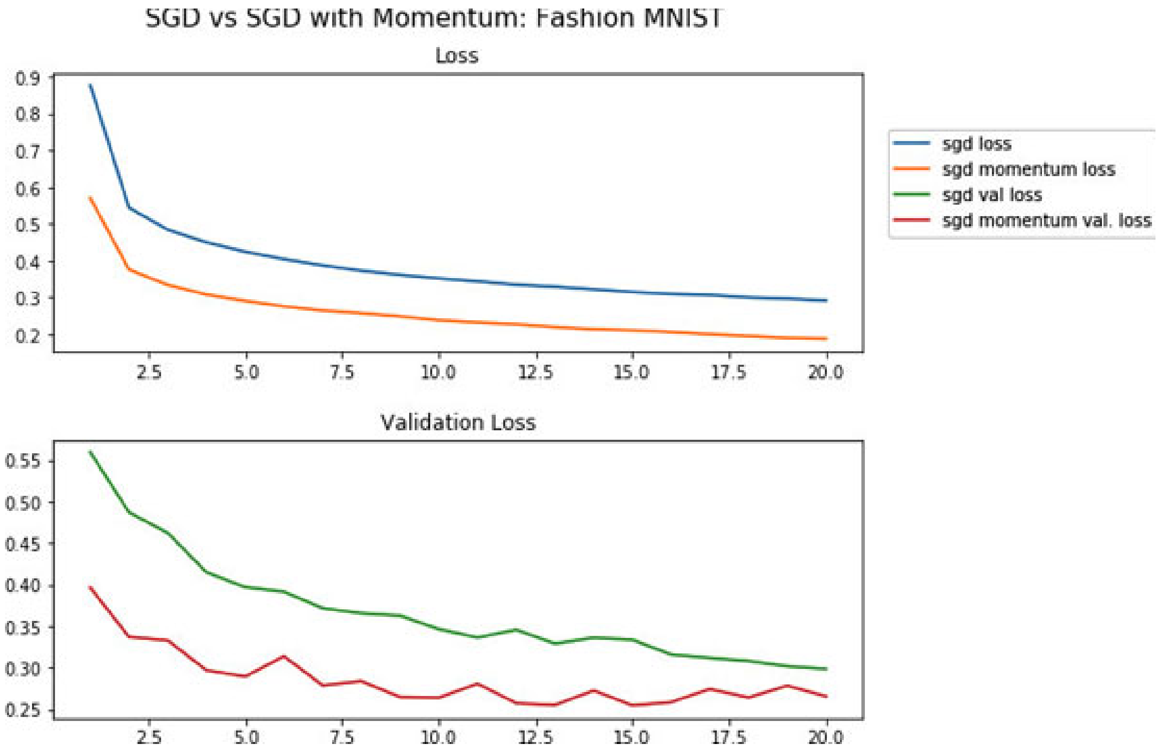


Figure 3.5: *Comparing training and validation loss for SGD and SGD with momentum*

Next, let us look at the **Nesterov** based gradient descent which is another variation of SGD, and supported by TensorFlow 2.x out of the box.

- **Nesterov based gradient descent**: Problem with momentum-based SGD is that when the local minima is reached, momentum is pretty high, hence the minima could be missed. This was noticed by a researcher called *Yuri Nesterov*. He proposed a research paper in *1983,* which is now called **Nesterov Accelerated Gradient**.

  Nesterov Accelerated Gradient (**NAG**) provides a way to give the momentum term this kind of prescience. We know that we will use the momentum term γv_{t-1} to move the parameters θ. Computing $\theta - \gamma v_{t-1}$ gives us an approximation of the next position of the parameters, which gives us a rough idea where the parameters are going to be. We can effectively look ahead by calculating the gradient not with respect to our current parameters θ but with respect to the approximate future position of our parameters:

  The following equation explains how parameters θ are calculated based on v_t

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Let us apply Nesterov accelerated gradient to Fashion MNIST:

```
model_sgd_m_n = get_model()
optimizer_sgd_m_nestrov=
tf.keras.optimizers.SGD(momentum=0.9,
 nesterov=True)
model_sgd_m_n.compile(optimizer=optimizer_sgd_m_nestrov ,
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
```

Plot the loss and validation loss, comparing it with momentum-based optimization. We have plotted this in the following *figure 3.6*, where you can see that Nesterov based loss is lower than normal loss:

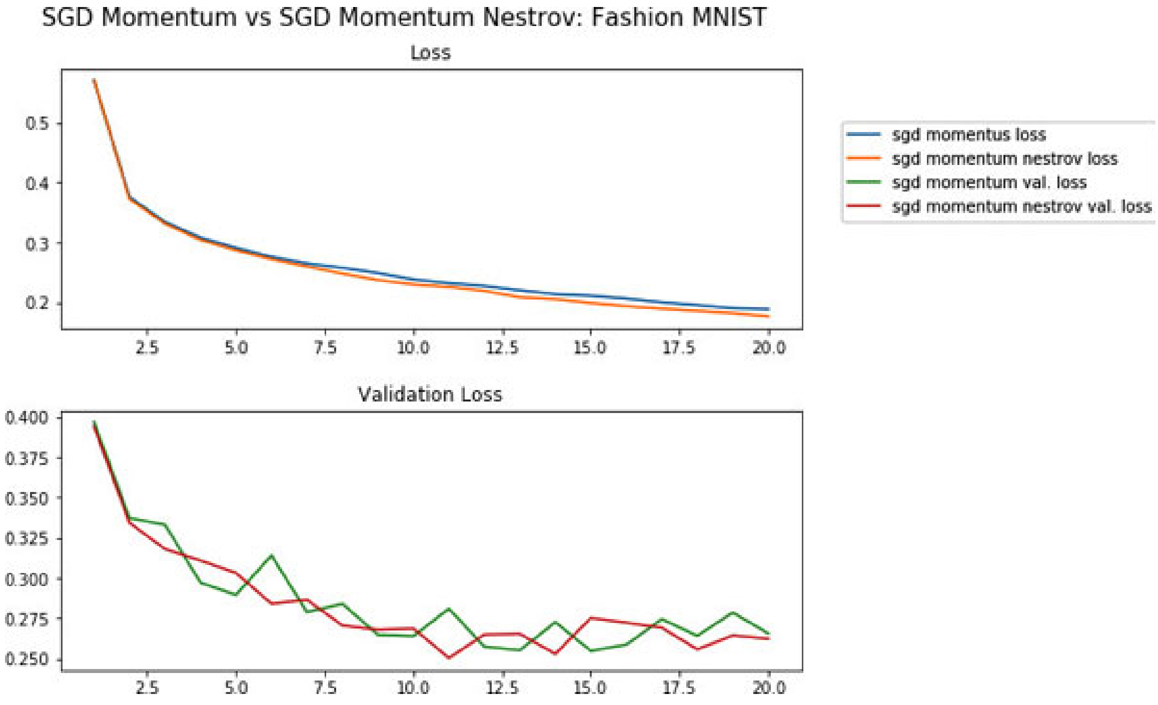


Figure 3.6: *SGD momentum versus SGD momentum with Nesterov*

As you can see, the loss is reduced for Nesterov-based technique, but the change is not significant. Let us compare the validation loss for both the techniques as well.

- **Adam optimizer**: Adam (which stands for **Adaptive Moment Estimation**) is another technique that computes adaptive learning rates for parameters. In addition to storing an average of past squared gradients (like **adadelta** which we explored earlier), it also keeps track of exponentially decaying average of past gradients m_t - very similar to momentum:

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\widehat{v}_t = \frac{v_t}{1-\beta_2^t}$$

The formula for the first moment is (m_t) and the second moment is (the variance - v_t) of the gradients.

Let us look at how the loss and validation loss for **Adam based optimizer** compare to SGD momentum-based optimizer. *Figure 3.7* shows the training and validation loss for SGD and Adam-based optimizer:

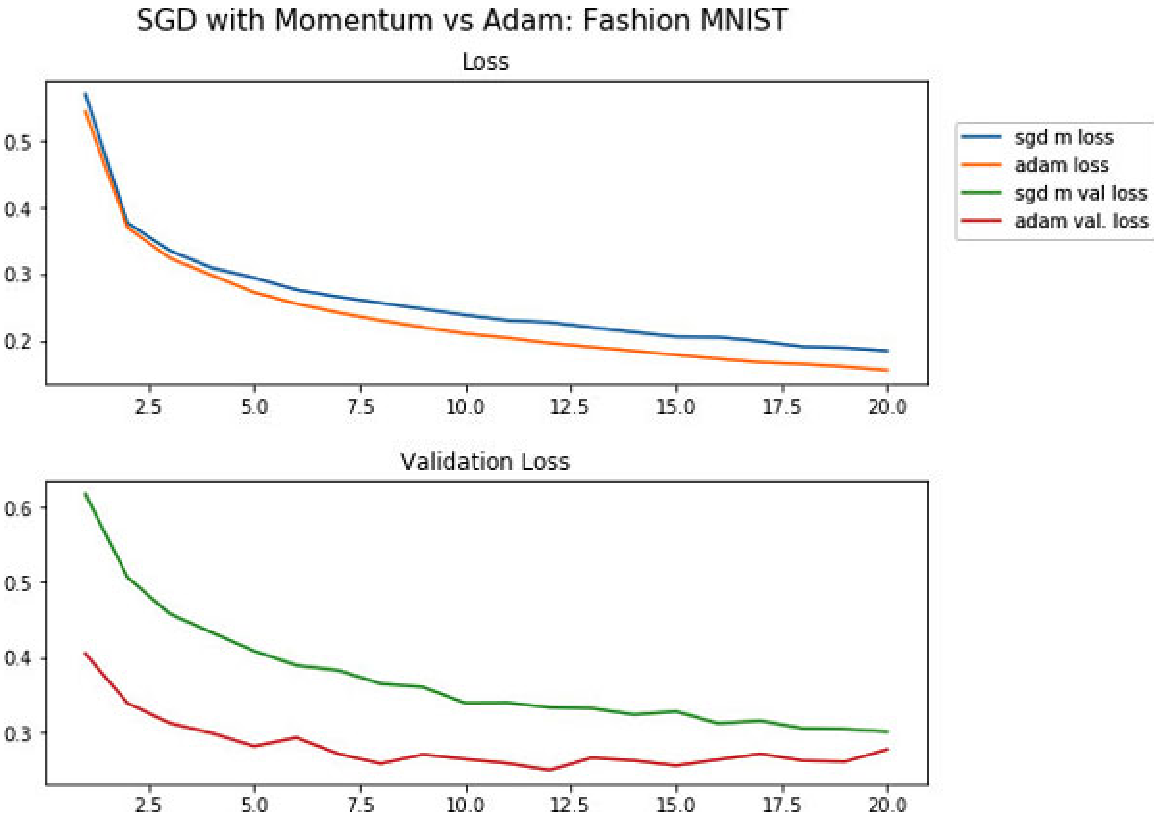


Figure 3.7: *SGD momentum versus Adam based training and validation loss*

There is minimal difference in the training loss, but the validation loss for Adam is quite low as compared to SGD–momentum based.

- **Adagrad**: Adagrad updates the learning rate based on parameters. It makes large updates on infrequent parameters and smaller changes to more frequently changing parameters. This optimizer uses a different learning rate for every parameter (η) at a time based on previous gradients for that parameter. Adagrad uses different learning rate for each parameter $\theta(i)$ for time step *t*. $g(t, i)$ is defined as a gradient of loss function for parameter $\theta(i)$at time step *t*. The following equation explains how each parameter is updated as a function of learning rate η:

$$\theta_{t+1}=\theta_t-\frac{\eta}{\sqrt{G_{t,ii}+\varepsilon}}\odot g_{t,i}$$

The key of this algorithm is in the matrix *Gt*, which is the sum of the outer product of the gradients until time-step *t*, which is defined as follows:

$$G=\sum_{t=1}^{t} g_t g_t^T$$

Adagrad in *TensorFlow 2.x* is implemented using the following function:

```
tf.keras.optimizers.Adagrad(
 learning_rate=0.001, initial_accumulator_value=0.1,
 epsilon=1e-07,
 name='Adagrad', **kwargs
)
Let us use it in Fashion MNIST classification use case:
model_adagrad = get_model()
model_adagrad.compile(optimizer='adagrad',
     loss='sparse_categorical_crossentropy',
     metrics=['accuracy'])
EPOCHS = 20
history_adagrad = model_adagrad.fit(X_train, y_train,
epochs=EPOCHS,
      validation_data=(X_test, y_test))
```

On plotting the training and validation loss for **Adagrad,** compared to **Adam,** we get the following plots as shown in the *figure 3.8*:

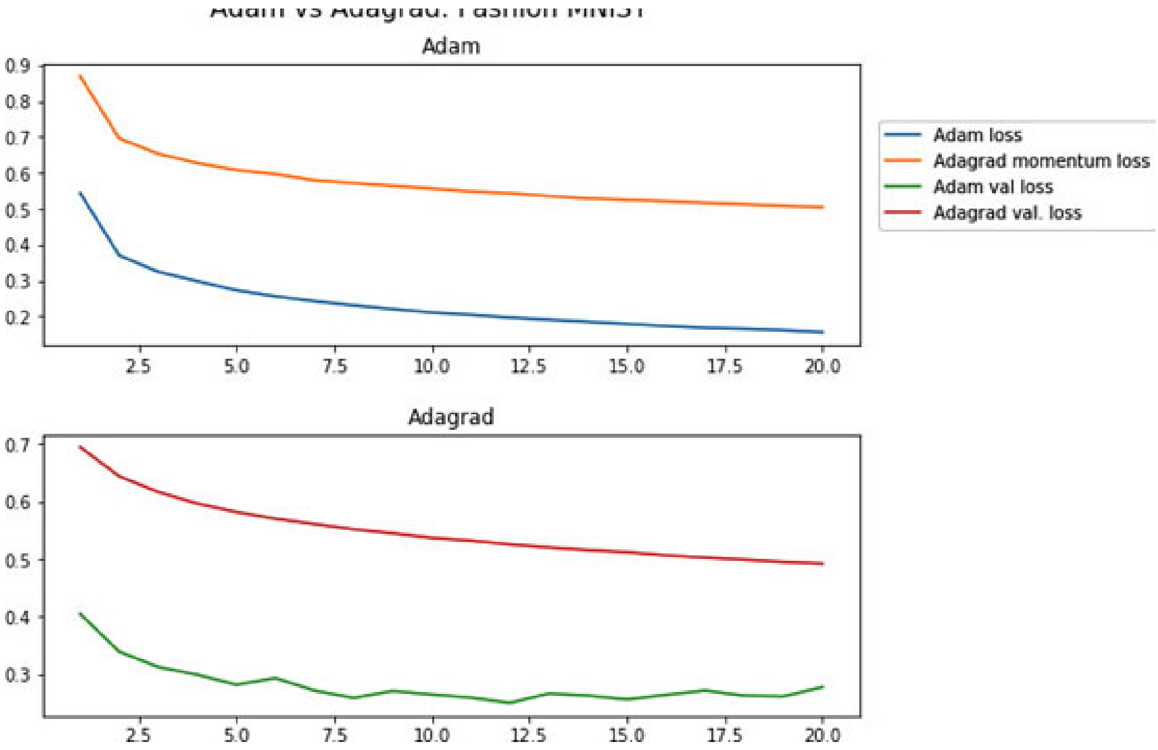


Figure 3.8: *Adam versus Adagrad loss*

The preceding plots clearly state that Adagrad is not a better choice as compared to Adam.

- **Adadelta**: Next, look at another variation called **Adadelta**. Adadelta is the extension of Adagrad to remove the decaying learning rate. Adadelta limits the window of past gradients to fixed size *w*. It stores the sum or gradients by recursively defining it as a decaying mean of all past squared gradients instead of storing all the previous gradients. The average $E[g^2](t)$ at time step *t* depends on previous average and current gradient:

$$E\left[g^2\right](t)=\gamma.E\left[g^2\right](t-1)+(1-\gamma).g^2(t)$$

Where γ to a similar value as the momentum term, around `0.9`.

So, the gradient *t+1* is calculated as follows:

$$\theta(t+1)=\theta(t)+\Delta\theta(t)$$

$$\Delta\theta(t)=-\eta g(t,i)$$

$$\Delta\theta(t)=-\frac{\eta}{\sqrt{E(g^{2}{}_{t})+\varepsilon}}g(t)$$

In TensorFlow, Adadelta is implemented by the following function:

```
tf.keras.optimizers.Adadelta(
  learning_rate=0.001, rho=0.95, epsilon=1e-07,
  name='Adadelta', **kwargs
)
```

Let us compare Adadelta's training and validation loss as compared to Adam, as shown in the *figure 3.9*:

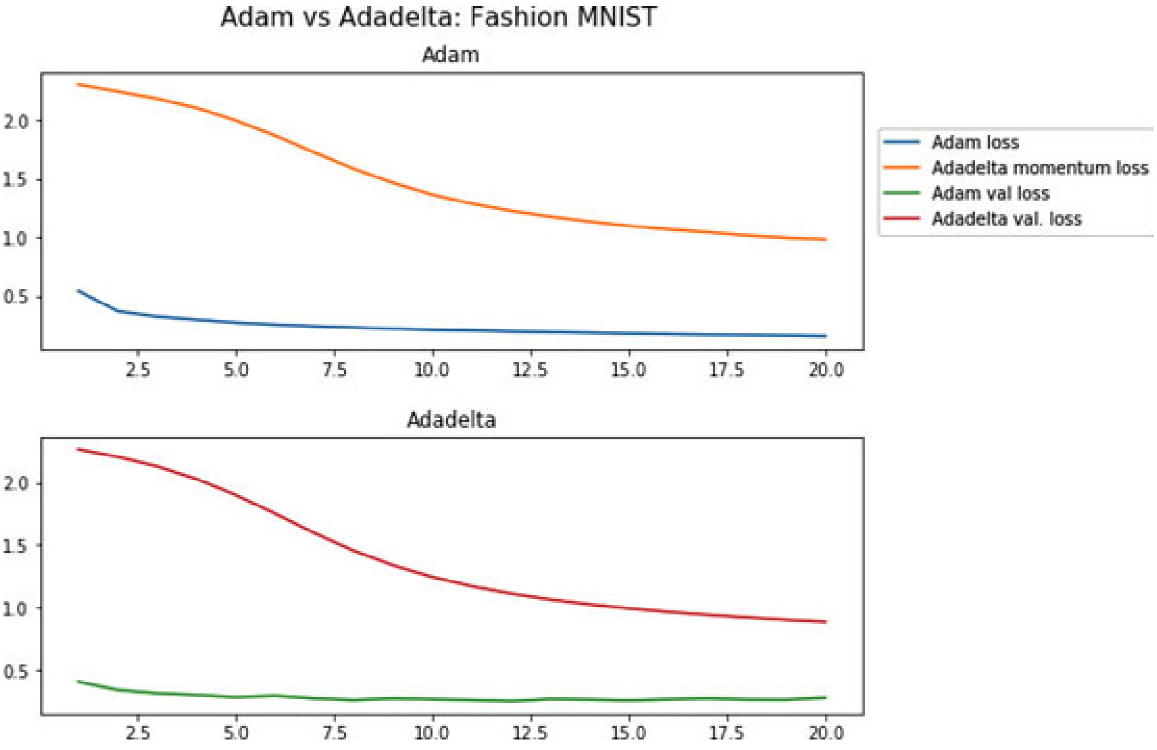


Figure 3.9: *Comparing Adam and Adadelta's training and validation loss*

We covered a few optimizers, but TensorFlow has support for others which we might not have covered here. Please refer to the following section.

Other optimizers

There are other optimizers that are supported by TensorFlow, which you can explore on your own.

This complete list of optimizers is supported by *TensorFlow 2.x* APIs:

- **class Adadelta**: Optimizer that implements the Adadelta algorithm. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adadelta
- **class Adagrad**: Optimizer that implements the Adagrad algorithm. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adagrad
- **class Adam**: Optimizer that implements the Adam algorithm. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam
- **class Adamax**: Optimizer that implements the Adamax algorithm. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adamax
- **class Ftrl**: Optimizer that implements the FTRL algorithm. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Ftrl
- **class Nadam**: Optimizer that implements the Nadam algorithm. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Nadam
- **class Optimizer**: Updated base class for optimizers. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Optimizer

- **class RMSprop**: Optimizer that implements the RMSprop algorithm. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/RMSprop
- **class SGD**: Stochastic gradient descent and momentum optimizer. For more information, please visit the following link:

  https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD

Now that we have a good idea about losses and optimizers, let us look at layers and models in more detail in the next section.

Layers and models using Keras functional APIs

Sequential APIs allow you to create basic models with layers in a sequence. If you want to create a more complex topology, *TensorFlow 2.x* supports Keras functional APIs that can be used to create multi-input, multi-output, and sharing of layers. Layer is considered a **callable object,** and passed on as a **function**.

The following code snippet shows how we can define a Keras model using the functional interface: the model is a fully connected neural network that accepts a *100-dimensional input* and produces a single number as output:

```
import tensorflow as tf
input_shape = (100,)
inputs = tf.keras.layers.Input(input_shape)
net = tf.keras.layers.Dense(units=64, activation=tf.nn.elu,
name="fc1")
  (inputs)
net = tf.keras.layers.Dense(units=64, activation=tf.nn.elu,
name="fc2")
  (net)
net = tf.keras.layers.Dense(units=1, name="G")(net)
model = tf.keras.Model(inputs=inputs, outputs=net)
```

Let us look at a full-fledged sample that regenerates MNIST images using a functional model.

Full code listing can be found at **https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch03/functional_apis_basic_1.ipynb**

1. Start by importing the relevant functions:

```
from tensorflow.keras import layers
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import UpSampling2D
from tensorflow.keras import Model
from tensorflow.keras.datasets import mnist
from tensorflow import keras
from tensorflow.keras import utils
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Let us first load the data and normalize:

```
nb_classes = 10
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.astype("float32")/255.
X_test = X_test.astype("float32")/255.
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
print(y_train.shape)
```

3. Categorical variables need to be converted to bin:

   from tensorflow.keras, import utils.

   utils.to_categorical: Converts a class vector (integers) to binary class matrix. API signature is listed as follows:

   keras.utils.to_categorical(y, num_classes=None, dtype='float32')

   Converts a class vector (integers) to a binary class matrix:

```
y_train = utils.to_categorical(y_train, nb_classes)
y_test = utils.to_categorical(y_test, nb_classes)
```

4. Reshape `X_train` and `X_test`:

```
import numpy as np
X_train = X_train.reshape((len(X_train),
np.prod(X_train.shape[1:])))
X_test = X_test.reshape((len(X_test),
np.prod(X_test.shape[1:])))
#len(X_train)
```

```
print(X_train.shape)
print(X_test.shape)
```

Print output will show the shapes:

```
(60000, 784)
(10000, 784)
```

Let us define a model, call it **`autoencoder`** using Keras functional APIs. It takes input, fed into 64 neuron hidden layer. Output layer is of the same dimension as input layer **`784`**:

```
input_size = 784
hidden_size = 64
output_size = 784
x = Input(shape=(input_size,))
h = Dense(hidden_size, activation='relu')(x)
r = Dense(output_size, activation='sigmoid')(h)
autoencoder = Model(inputs=x, outputs=r)
autoencoder.compile(optimizer='adam', loss='mse')
```

5. Let us train the model:

```
epochs = 5
batch_size = 128
history = autoencoder.fit(X_train, X_train,
batch_size=batch_size, epochs=epochs, verbose=1,
       validation_data=(X_test, X_test))
```

Output will show the loss for each of the five iterations:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/5
60000/60000 [==============================] - 2s
28us/sample - loss: 0.0454 - val_loss: 0.0231
Epoch 2/5
60000/60000 [==============================] - 1s
20us/sample - loss: 0.0177 - val_loss: 0.0131
Epoch 3/5
60000/60000 [==============================] - 1s
19us/sample - loss: 0.0111 - val_loss: 0.0090
Epoch 4/5
60000/60000 [==============================] - 1s
19us/sample - loss: 0.0080 - val_loss: 0.0068
Epoch 5/5
```

```
60000/60000 [==============================] - 1s
20us/sample - loss: 0.0064 - val_loss: 0.0057
```

6. We create another model, which has no hidden layer, by reusing the **x** and **h** layers defined and populated by training call:

```
conv_encoder = Model(x, h)
```

Predict the encoded images:

```
encoded_imgs = conv_encoder.predict(X_test)
```

7. Let us plot the encoded images:

```
n = 10
plt.figure(figsize=(20, 8))
for i in range(n):
   ax = plt.subplot(1, n, i+1)
   plt.imshow(encoded_imgs[i].reshape(4, 16).T)
   plt.gray()
   ax.get_xaxis().set_visible(False)
   ax.get_yaxis().set_visible(False)
plt.show()
```

Output will be a *10x10* grid of encoded images as shown in *figure 3.10*:

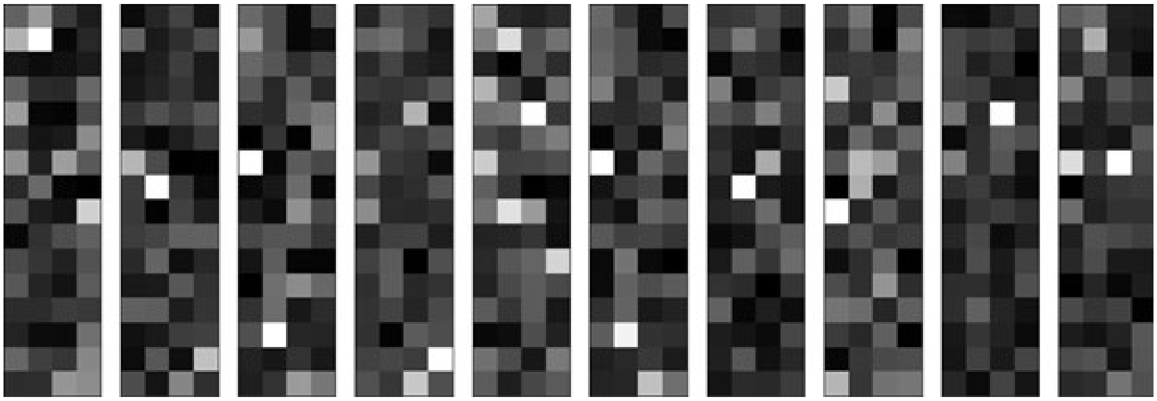


Figure 3.10: *Encoded images created by conv_encoder*

Next, let us use the **autoencoder** to recreate the images and plot them:

```
decoded_imgs = autoencoder.predict(X_test)
n = 10
plt.figure(figsize=(20, 6))
for i in range(n):
   # display original
   ax = plt.subplot(3, n, i+1)
   plt.imshow(X_test[i].reshape(28, 28))
```

```
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # display reconstruction
    ax = plt.subplot(3, n, i+n+1)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Output of the plot will show the reconstructed images alongside the original ones as shown in *figure 3.10.1*:

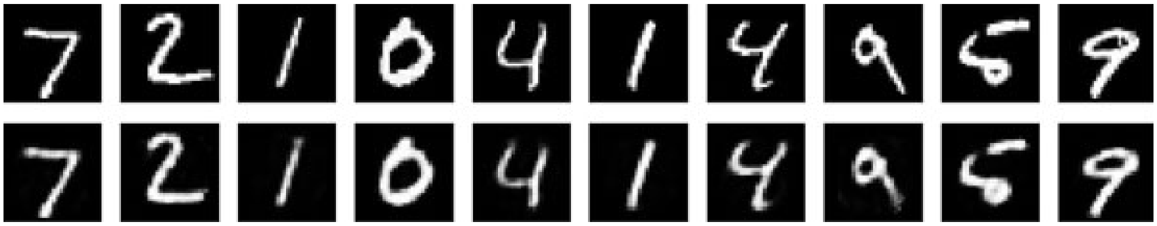


Figure 3.10.1: *Reconstructed versus original images*

As can be seen, the reconstruction is quite close to the original images. Let us look at the training and validation loss as shown in *figure 3.11*:

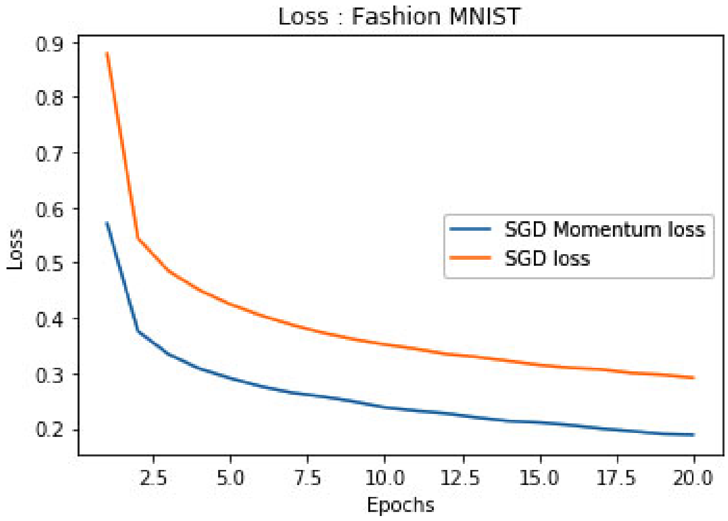


Figure 3.11: *Training and validation loss for autoencoder*

The validation and training losses converge very well giving the visually accurate reconstructed images.

Effective TensorFlow 2.x

In this section, we look at how to work more effectively with TensorFlow, using concepts like **namespaces**. We will also look at the changes made in *TensorFlow 2.x* in more detail.

Reorganization of namespaces

TensorFlow symbols are defined in terms of logical namespaces. These namespaces have been reorganized in the new library. Endpoint in TensorFlow is defined as a full name to access the resource. For example, **`tf.name_scope`** has two endpoints.

```
 tf.name_scope and tf.keras.backend.name_scope.
```

The goal of TensorFlow 2.x is to re-organize the namespaces to fewer ones, and club logical endpoints together. New namespaces that have been added are listed as follows:

- **`tf.random`**: contains random sampling ops
- **`tf.keras.layers`**: contains all symbols that are currently under **`tf.layers`**
- **`tf.keras.losses`**: contains all symbols previously under **`tf.losses`**
- **`tf.keras.metrics`**: contains all symbols previously under **`tf.metrics`**
- **`tf.debugging`**: ops for debugging, such as asserts. TensorFlow debugger also has been moved under **`tf.debugging`** namespace
- **`tf.dtypes`**: data types
- **`tf.io`**: ops for reading and writing
- **`tf.quantization`**: ops related to quantization

New symbols have been migrated from **`tf.layers`**, **`tf.losses`**, **`tf.metrics`** to **`tf.keras.layers`**, **`tf.keras.losses`**, and **`tf.keras.metrics`**.

There are also additional endpoints added as listed in *Reference [3.1]*.

Deprecated APIs

These endpoints have been removed:

- **`tf.logging`**: Python logging module to be used instead.
- **`tf.manip`**: We will keep endpoints in root for symbols in **`t`**.

Eager execution

As we learnt in *The Sequential API* section, eager execution allows you to use TensorFlow as a standard Python library that is executed immediately by the Python interpreter.

Baseline example:

```
import tensorflow.compat.v1 as tf tf.disable_v2_behavior()
A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
x = tf.constant([[0, 10], [0, 0.5]])
b = tf.constant([[1, -1]], dtype=tf.float32)
y = tf.add(tf.matmul(A, x), b, name="result") #y = Ax + b
with tf.compat.v1.Session() as sess:
  print(sess.run(y))
```

Same code with TensorFlow 2.x is:

```
import tensorflow as tf
A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
x = tf.constant([[0, 10], [0, 0.5]])
b = tf.constant([[1, -1]], dtype=tf.float32)
y = tf.add(tf.matmul(A, x), b, name="result")
print(y)
```

No need to worry about the graph and session.

In *TensorFlow 1.x*, a **`tf.Tensor`** object was only a symbolic representation of the output of a **`tf.Operation`**; in 2.x, this is not the case.

Since the operations are executed as soon as the Python interpreter evaluates them, every **`tf.Tensor`** object is not only a symbolic representation of the output of a **`tf.Operation`**, but also a concrete Python object that contains the result of the operation.

Since a **`tf.Tensor`** object is still a symbolic representation of the output of a **`tf.Operation`**, it allows it to support and use 1.x features in order to manipulate **`tf.Tensor`** objects. Thereby, building graphs of **`tf.Operation`** that produce **`tf.Tensor`**.

The graph is still present and the **`tf.Tensor`** objects are returned as a result of every TensorFlow method.

TensorFlow 2.x, with eager execution, allows the user to design better-engineered software. In *1.x version*, TensorFlow had the omnipresent concepts of global variables, collections, and sessions.

Variables and collections could be accessed from everywhere in the source code since a default graph was always present.

The session is required in order to organize the complete project structure, since it knows that only a single session can be present. Every time a node had to be evaluated, the session object had to be instantiated and accessible in the current scope.

TensorFlow 2.x changed all of these aspects, increasing the overall quality of code that can be written using it. In practice, before 2.x, using TensorFlow to design a complex software system was tough, and many users just gave up and defined huge single file projects that had everything inside them. Now, it is possible to design software in a better and cleaner way by following all the software engineering's good practices.

Functions replace sessions

The `tf.Session` object has been removed from the TensorFlow API. By using eager execution, you no longer need the session because the execution of the operation is immediate—we don't need to build a computational graph before running the computation.

This means the source code can be organized better. In *TensorFlow 1.x*, it was difficult to design software by following object-oriented principles, or to create modular code that used Python functions. In *TensorFlow 2.x*, this is easier and highly recommended.

Removing globals

API implementation of TensorFlow variables had the following drawbacks: difficult to understand semantics and reliance on global scopes and global collections. *TensorFlow 2.x* API is moving towards becoming more pythonic and object-oriented the more infrastructure for global variables is being deprecated.

The main changes in the variables are as follows:

- `tf.Variable` has become an abstract base class with an interface and a scoped factory method for creating instances.
- Users can create their own instances of `tf.Variable` by sub classing.
- If a variable can be shared across sessions, processes will be controlled by a constructor argument to `tf.Variable`. No other type of scope reuse

has been done in the framework.

- Libraries and users are being encouraged to reuse variables by reusing their objects, like Keras layers implementation does.

Let us look at a simple example of how globals used to work in 1.x. First, we will define a program that calculates **`y`** from **`A, x, b`** and **`z`**:

```
import tensorflow as tf
def get_y():
 A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32, name="A")
 x = tf.constant([[0, 10], [0, 0.5]], name="x")
 b = tf.constant([[1, -1]], dtype=tf.float32, name="b")
 # I don't know what z is: if is a constant or a variable
 z = tf.get_default_graph().get_tensor_by_name("z:0")
 y = A @ x + b - z
 return y
#test = tf.Variable(10., name="z")
#del test
#test = tf.constant(10, name="z")
#del test
y = get_y()
with tf.Session() as sess:
 init = tf.global_variables_initializer()
 sess.run(init)
 print(sess.run(y))
```

Since we never defined **`z`**, we get an error.

KeyError: "The name 'z:0' refers to a Tensor which does not exist. The operation, 'z', does not exist in the graph".

Let us create a variable **`z`**, and rerun the function:

```
import tensorflow as tf
def get_y():
  A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32, name="A")
  x = tf.constant([[0, 10], [0, 0.5]], name="x")
  b = tf.constant([[1, -1]], dtype=tf.float32, name="b")
  # I don't know what z is: if is a constant or a variable
  z = tf.get_default_graph().get_tensor_by_name("z:0")
  y = A @ x + b - z
  return y
test = tf.Variable(10., name="z")
```

```
y = get_y()
with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    print(sess.run(y))
We get the output:
[[-9. 0.]
[-9. 21.]]
```

Even if we add or delete after variable definition, and rerun the cell without restarting the Jupyter kernel, the program will still work since **`z`** is already initialized in the TensorFlow graph and is a **global**:

```
import tensorflow as tf
def get_y():
  A = tf.constant([[1, 2], [3, 4]], dtype=tf.float32, name="A")
  x = tf.constant([[0, 10], [0, 0.5]], name="x")
  b = tf.constant([[1, -1]], dtype=tf.float32, name="b")
  # I don't know what z is: if is a constant or a variable
  z = tf.get_default_graph().get_tensor_by_name("z:0")
  y = A @ x + b - z
  return y
y = get_y()
with tf.Session() as sess:
  init = tf.global_variables_initializer()
  sess.run(init)
    print(sess.run(y))
```

These are some of the problems that TensorFlow 2.x tries to remove by removing **`global`**.

Control flow

Executing sequential operations in *TensorFlow 1.x* is not an easy task if the operations had no explicit order of execution constraints. For example, if we want to use TensorFlow to do the following:

1. Declare and initialize two variables: **`x`** and **`y`**.
2. Increase the value of **`y`** by **`1`**.
3. Compute **`x*y`**.
4. Repeat this five times.

The first non-working attempt in *TensorFlow 1.x* is to just declare the code by following the preceding steps:

```
import tensorflow as tf
x = tf.Variable(1, dtype=tf.int32)
y = tf.Variable(2, dtype=tf.int32)
assign_op = tf.assign_add(y, 1)
out = x * y
init = tf.global_variables_initializer()
with tf.Session() as sess:
   sess.run(init)
   for _ in range(5):
    print(sess.run(out))
```

```
1
1
1
1
1
```

Output will be only **2** since out is never reevaluated. To fix it we have to use:

```
with tf.control_dependencies([assign_op]):
   out = x * y
init = tf.global_variables_initializer()
with tf.Session() as sess:
   sess.run(init)
   for _ in range(5):
   print(sess.run(out))
```

This fixes the problem as can be seen in the output as follows:

```
2
3
4
5
6
```

TensorFlow 2.x, with its eager execution, makes it possible to use the Python interpreter to control the flow of execution:

```
import tensorflow as tf
print(tf.__version__)
x = tf.Variable(1, dtype=tf.int32)
y = tf.Variable(1, dtype=tf.int32)
for _ in range(5):
```

```
y.assign_add(1)
out = x * y
print(out)
```

Combine tf.data.Dataset and tf.function

When iterating over training data that fits into memory, you can use regular Python iteration, else `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) is the most effective way to stream training data from disk. Datasets are iterables, and work just like other Python iterables in the *Eager* mode. You can utilize dataset async prefetching/streaming features by wrapping your code in `tf.function()`, which will replace Python iteration with the equivalent graph operations using **AutoGraph**.

1. Let us look at a simple example of how this can be implemented.

   Code reference: tensorflow_2.x_book_code/ch03/custom_TF_training_loops.ipynb.

   ```
   import tensorflow as tf
   import numpy as np
   (X_train, y_train), (X_test, y_test) =
   tf.keras.datasets.fashion_mnist.load_data()
   X_train = X_train/255.
   X_test = X_test/255.
   X_train.shape, X_test.shape, y_train.shape, y_test.shape
   # Define the labels of the dataset CLASSES=["T
   shirt/top","Trouser","Pullover","Dress","Coat","Sandal","Shi
   rt","Sneaker","Bag","Ankle boot"]
   # Change the pixel values to float32 and reshape input data
   X_train = X_train.astype("float32").reshape(-1, 28, 28, 1)
   X_test = X_test.astype("float32").reshape(-1, 28, 28, 1)
   ```

2. Create a sequential training model:

   ```
   # TensorFlow imports
   from tensorflow.keras.models import *
   from tensorflow.keras.layers import *
   # Define utility function for building a basic shallow
   Convnet
   def get_training_model():
     model = Sequential()
   ```

```
    model.add(Conv2D(16, (5, 5), activation="relu",
    input_shape=(28, 28,1)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(32, (5, 5), activation="relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation="relu"))
    model.add(Dense(len(CLASSES), activation="softmax"))
    return model
```

3. Define loss function and optimizer:

```
# Define loss function and optimizer
loss_func = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()
```

   Calculate the average loss across the batch size within an epoch and specify the performance metric.

```
# Average the loss across the batch size within an epoch
train_loss = tf.keras.metrics.Mean(name="train_loss")
valid_loss = tf.keras.metrics.Mean(name="test_loss") #
Specify the performance metric train_acc =
tf.keras.metrics.SparseCategoricalAccuracy(name="train_acc")
valid_acc =
tf.keras.metrics.SparseCategoricalAccuracy(name="valid_acc")
```

4. Train the model by calculating the gradient tape in a callable function. Notice how this is annotated with **`tf.function`**:

```
@tf.function
def model_train(features, labels):
    # Define the GradientTape context
    with tf.GradientTape() as tape:
    # Get the probabilities
    predictions = model(features)
    # Calculate the loss
    loss = loss_func(labels, predictions)
    # Get the gradients
    gradients = tape.gradient(loss, model.trainable_variables)
    # Update the weights
    optimizer.apply_gradients(zip(gradients,
    model.trainable_variables))
    # Update the loss and accuracy
   train_loss(loss)
   train_acc(labels, predictions)
```

5. Create a callable function to validate the model:

```
@tf.function
def model_validate(features, labels):
    predictions = model(features)
    v_loss = loss_func(labels, predictions)
  valid_loss(v_loss)
    valid_acc(labels, predictions)
```

6. Instantiate the model by calling **`get_training_model()`**:

```
model = get_training_model()
```

7. Grab random images from the test and make predictions using the model while it is training and log them:

```
def get_sample_predictions():
    predictions = []
    images = []
    random_indices = np.random.choice(X_test.shape[0], 25)
    for index in random_indices:
    image = X_test[index].reshape(1, 28, 28, 1)
    prediction = np.argmax(model(image).numpy(), axis=1)
    prediction = CLASSES[int(prediction)]
    images.append(image)
    predictions.append(prediction)
```

Putting it all together:

```
for epoch in range(5):
    # Run the model through train and test sets respectively
    for (features, labels) in train_ds:
    model_train(features, labels)
    for test_features, test_labels in test_ds:
    model_validate(test_features, test_labels)
    # Grab the results
    (loss, acc) = train_loss.result(), train_acc.result()
    (val_loss, val_acc) = valid_loss.result(),
    valid_acc.result()
    # Clear the current state of the metrics
    train_loss.reset_states(), train_acc.reset_states()
    valid_loss.reset_states(), valid_acc.reset_states()
```

```
  # Local logging
template = "Epoch {}, loss: {:.3f}, acc: {:.3f}, val_loss:
{:.3f}, val_acc: {:.3f}"
  print (template.format(epoch+1,
      loss,
      acc,
      val_loss,
      val_acc))
 get_sample_predictions()
```

You will see the output that shows the loss and accuracy of 5 epochs:

```
Epoch 1, loss: 0.558, acc: 0.797, val_loss: 0.435, val_acc:
0.844
Epoch 2, loss: 0.361, acc: 0.869, val_loss: 0.361, val_acc:
0.872
Epoch 3, loss: 0.309, acc: 0.888, val_loss: 0.344, val_acc:
0.874
Epoch 4, loss: 0.278, acc: 0.899, val_loss: 0.334, val_acc:
0.876
Epoch 5, loss: 0.254, acc: 0.907, val_loss: 0.306, val_acc:
0.887
```

Notice how difficult it is to implement as compared to model.fit, which does this behind the scene.

Conclusion

In this chapter, we did a deep dive into Keras functional APIs, and learnt why it became so popular. We learnt how loss functions and optimizers are implemented in these APIs. We learnt how to write effective TensorFlow applications with 2.x based functional APIs and the APIs which have been deprecated. We also learnt about how various optimizers in Keras fare against Fashion MNIST. In the end, we learnt how Keras overcomes problems in programming against TensorFlow 1.x APIs. In the next chapter, we look at a special type of neural network called **convolutional neural network,** which utilizes spatial information to extract features from an image, video, or text.

Questions

1. **What has replaced Sessions in 2.x?**

a. functions
b. not replaceable
c. none of the above

2. **Are global scopes supported in TensorFlow 2.x?**

a. true
b. false

3. **Which of the following APIs are deprecated in TensorFlow 2.x?**

a. `tf.logging` and `tf.manip`.
b. `tf.logging`.
c. `tf.manip`.

Code listing

- Using training and evaluation loop:
  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch03/training_and_evaluation_fashionmnist.ipynb
- Loss, metrics, and an optimizer:
  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch03/losses.ipynb
- Optimizers:
  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch03/multiple_optimizers.ipynb
- Layers and models using Keras functional APIs:
  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch03/functional_apis_basic_1.ipynb

References

1. https://github.com/tensorflow/community/blob/master/rfcs/20180827-api-names.md.
2. https://github.com/tensorflow/community/pull/11.
3. https://www.tensorflow.org/beta/guide/effective_tf2.

CHAPTER 4
Convolutional Neural Networks

Introduction

Convolutional Neural Network (**CNN**) are a type of neural networks that specialize in processing images, videos, and texts. They provide tools for leveraging spatial information. For example, in an image, pixels which are close together also vary in the same way. Convolutional networks learn to exploit the covariance between pixels to learn more effectively and efficiently. In the following sections, we go into more details on using *TensorFlow 2.6*.

Structure

In this chapter, we will cover the following topics:

- Introduction of convolutional networks
- Building your first ConvNet
- Transfer learning with TensorFlow 2.6
- VGG16, VGG19, Inception V3
- Introducing **TensorFlow Hub** (**TFHub**)
- Leveraging TFHub for text processing

Objective

This chapter will introduce you to convolutional networks, its concept, and its implementation in TensorFlow. We start with a basic sample, and then move on to advanced concepts like transfer learning.

Introduction to convolutional networks

Convolutional networks are also known as **Convolutional Neural Networks** or **CNNs**. CNNs are a specific type of neural networks that process data in a grid-like topology. CNNs are used in many operations such as image retrieval, object detection, self-driving cars, semantic segmentation, text processing, face recognition, analyzing satellite data, image classification, and so on.

CNNs are made up of nodes (also called **neurons**) that have learnable weights and biases. CNN is a sequence of layers. In ConvNet, there are four main layers:

- Convolutional layer
- Flatten layer
- Pooling layer
- Fully-connected layer

ConvNets are very useful in image recognition and classification. ConvNet can be explained by using computer vision.

In image classification, let's consider an input image of size *128× 128,* and try to identify the object which is present in the picture, that is, *128× 128× 3,* because here, *3* denotes three channels of an RGB image. If we multiply the *128× 128× 3* then, we get *49152* (dimensions). But here, *128 by 128* is a very small image. So now, let's consider an image of size *1000× 1000× 3*, and if we multiply this product, then we get *3000000* (dimensions). That means, here we have three million input features. In the first layer of ConvNet, the output of neurons is connected to the local input region and performs dot product. In this layer, if we have *1000* hidden units, then the total number of weights of neurons is denoted by *W1 (matrix)*. This matrix in a standard, fully-connected network may be of size *1000 by 3* million. Then, the total dimension of the matrix may be up to *3 billion,* which is very large and therefore it is complicated to get the data to prevent the neural network from overfitting. Training the neural network with these parameters is not possible.

There are four main operations in the ConvNet:

- Convolution
- Flattening
- Non-linearity
- Pooling or sub-sampling
- Classification

These are the basic building blocks of every convolutional neural network. The name convolutional neural network means that the network topology contains nodes, which employs a mathematical operation called **convolution**. It is a specific kind of linear operation. Convolutional networks are neural networks that use convolution - a mathematical operation, in place of general matrix multiplication in at least one of their layers. Let us describe the mathematical operation of convolution. Then, we will discuss the concept of pooling and how it helps CNN.

Short comings of fully-connected networks

Fully connected networks have some inherent limitations in processing large tensors generated from datasets containing images. This leads to a very high training time and huge amount of in-memory footprint to run these networks.

CNNs offer a solution to the shortcomings discussed previously, though they work in a similar way like fully connected networks, except being more efficient.

CNNs handle **multidimensional data** (for example, images with RGB values have height, width, and depth in their tensor). CNN outputs are also three dimensional. Each neuron in CNN has access to only some neurons from the previous layer, referred to as a receptive field or filter size:

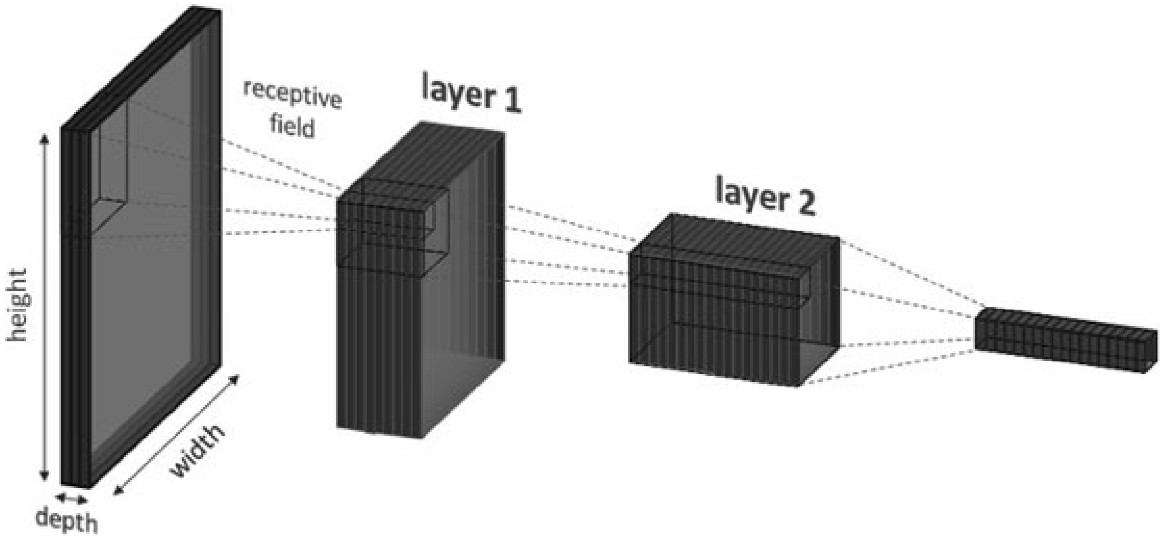


Figure 4.1: *Convolution filters being applied to a three-dimensional image tensor*

By linking neurons only to their neighboring ones in the previous layer, CNNs reduce the number of parameters to train while preserving the localization of image features.

Concepts

These neurons with shared weights and bias can also be considered as a single neuron sliding over the input matrix with spatially limited connectivity (limiting how neurons affect each other). At each step, this neuron is only spatially connected to the local region in the input volume (*H* × *W* × *D*) it is currently sliding over. Given this limited input of dimensions, $k_H \times k_W \times D$ for a neuron with a filter size (or kernel size) (k_H, k_W), the neuron still works like the one we talked about earlier. It linearly combines the input values ($k_H \times k_W \times D$ values) before applying an activation function to the sum (a linear or non-linear function). Mathematically, the response,$z_{i,j}$, of the neuron when presented with the input patch starting at position *(i, j)* can be expressed as follows:

$$z_{i,j} = \sigma\left(b + \overset{k_H-1}{\underset{l=0}{}} \overset{k_W-1}{\underset{m=0}{}} \overset{D\_1}{\underset{n=0}{}} w_{l,m,n} x_{i+l,j+m,n} \right)$$

where $\mathbf{w} \in \mathbb{R}^{k_H \times k_W \times D}$ are the weights with dimension $k_H \times k_W \times D$, $b \in \mathbb{R}$ is the bias and σ is the activation function.

By repeating this operation for each position, we obtain the complete response matrix *z* of dimensions *Ho* × *Wo, Ho* and *Wo* being the number of times a neuron can slide vertically and horizontally over the input tensor.

'Padding' in convolutional neural network is referred to the number of pixels added to an image before processing. As an example, if padding is set to zero, every pixel value added will be zero. If it is set to one, there will be one pixel added to the border of the image.

Padding extends the area that the convolutional neural network processes. It is a filter which moves across image, scanning pixels, and converting data into a smaller or larger format. Padding is added to the frame to assist kernel in processing the image. It allows more space for kernel to cover and a more accurate analysis of the image.

Let us look at an example of a convolution where a *3x3* filter is applied to an input of *3x3*. Notice the padding of *0* around the input *x*:

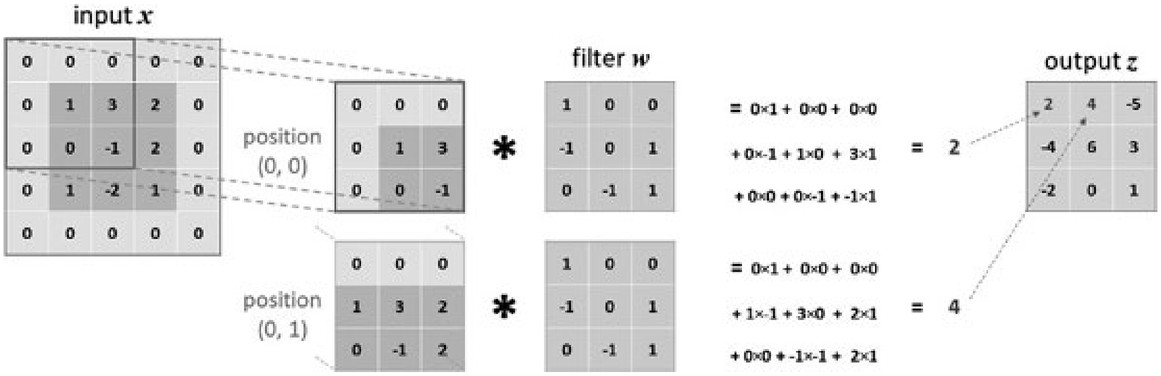


Figure 4.2: *Convolution of 3x3 filter on 3x3 matrix*

Convolutional operation is explained mathematically as follows:

$$(w * x)_{i,j} = \sum_{l=0}^{k-1}\sum_{m=0}^{k-1} w_{l,m} \cdot x_{i+l,j+m}$$

Convolution in TensorFlow 2.7

Available in the low-level API, `tf.nn.conv2d()` or `tf.keras.layers.Conv2D` are the default choices for image convolution. Some of the parameters are as follows:

- `input`: It is the batch of input images, of shape *(B, H, W, D)*, with *B* being the batch size.
- `filter`: It is the *N* filters stacked into a tensor of shape *(kH, kW, D, N)*.
- `strides`: It is a list of four integers representing the stride for each dimension of the batched input. Typically, you would use *[1, sH, sW, 1]* (that is, applying a custom stride only for the two spatial dimensions of the image).
- `padding`: It is either a list of *4 × 2* integers representing the padding before and after each dimension of the batched input, or a string defining which predefined padding case to use; that is, either **VALID** or **SAME** (explanation follows).
- `name`: It is the name to identify this operation (useful for creating clear, readable graphs).

Note that `tf.nn.conv2d()` accepts some other parameters, which we have not covered yet (refer to the documentation).

Sample code available at the following link: https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/discover_cnns_basic_ops.ipynb.

Let us look at the sample code, starting with reading the image. Sample takes care of both the scenarios -> running the code locally on Jupyter or Google Colab:

```
if IN_COLAB:
  image o char-style-override-39
  io.imread("https://github.com/rajdeepd/tensorflow_2.0_book_code/
  raw/master/ch04/res/peacock-in-black-and-white.jpg")
else:
  image o char-style-override-39 io.imread("./res/peacock-in-
  black-and-white.jpg")
```

The following figure shows how a simple convolution can be applied to an image:

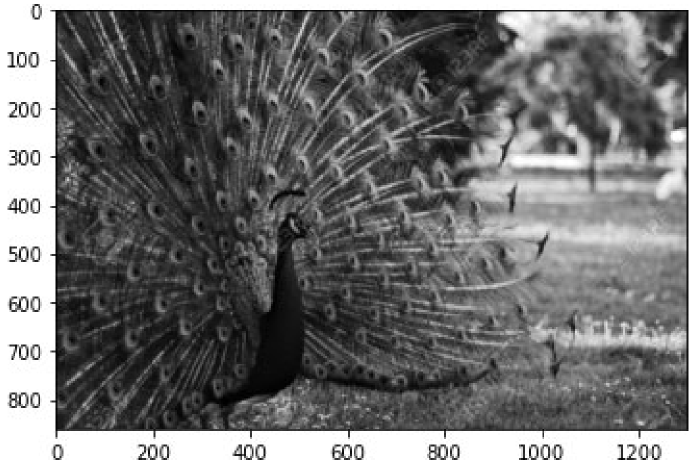


Figure 4.3: *Loaded image*

Raw image is loaded from GitHub or local directory. We converted it into a tensor and expanded the dimensions:

```
image o char-style-override-39 tf.convert_to_tensor(image,
tf.float32, nameo char-style-override-39"input_image")
image o char-style-override-39 tf.expand_dims(image, axiso char-
style-override-390) # we expand our tensor, adding a dimension at
position 0
print("Tensor shape: {}".format(image.shape))
```

`expand_dims` does not add or reduce elements in a tensor. This operation changes the shape by adding `1` to the dimensions. For example, a vector with `10` elements could be treated as a *10x1* matrix.

The use case to use expand_dims is when we have to build a ConvNet to classify grayscale images. The grayscale images are loaded as a matrix of size (as an example) *[320, 320]*. However, tf.nn.conv2d requires input to be *[batch, in_height, in_width, in_channels]*, where the in_channels dimension is missing in the data which, in this case, should be 1. So, we have to use expand_dims to add one more dimension.

Notice how the shape changes from tensor shape *(1, 861, 1300)* to *(1, 861, 1300, 1)*.

Let us define the convolution we are going to run on the preceding *figure 4.3*. It uses a *3x3* matrix and called a **Gaussian Blur**:

```
kernel o char-style-override-39 tf.constant([[1 / 16, 2 / 16, 1 /
16],
      [2 / 16, 4 / 16, 2 / 16],
      [1 / 16, 2 / 16, 1 / 16]], tf.float32,
      nameo char-style-override-39"gaussian_kernel")
kernel o char-style-override-39
tf.expand_dims(tf.expand_dims(kernel, axis=-1), axis=-1)
```

Apply the filter using **`tf.nn.conv2d`**:

```
blurred_image = tf.nn.conv2d(image, kernel, strideso =[1, 1, 1,
1], paddingo ="SAME")
```

Let us plot the image:

```
blurred_res = blurred_image.numpy()
# We "unbatch" our result by selecting the first (and only)
image; we also remove the depth dimension:
blurred_res = blurred_res[0, ..., 0]
plt.imshow(blurred_res, cmapo =plt.cm.gray)
```

Output of the plot is displayed in the following *figure 4.4*:

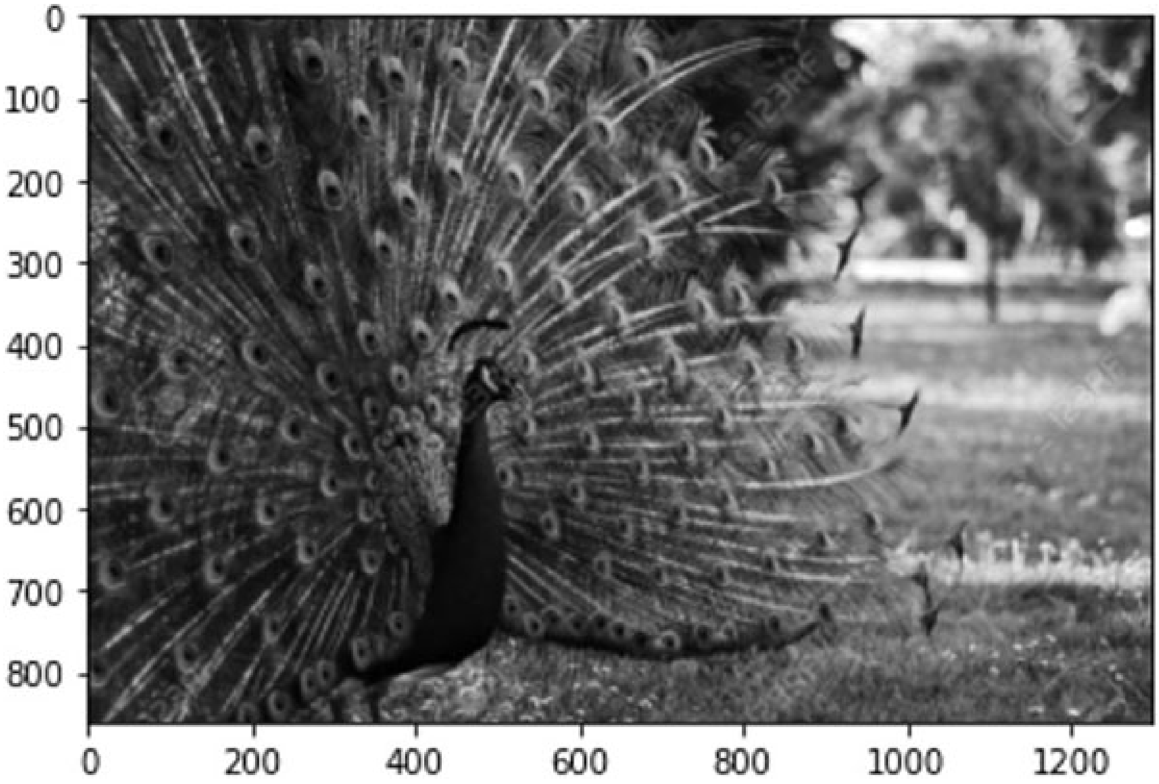


Figure 4.4: *Blur filter applied to the original image*

Next, let us apply the contour detection kernel:

```
kernel = tf.constant([[-1, -1, -1],
      [-1, 8, -1],
      [-1, -1, -1]], tf.float32, name="edge_kernel")
kernel = tf.expand_dims(tf.expand_dims(kernel, axis=-1), axis=-1)
edge_image = tf.nn.conv2d(image, kernel, strides=[1, 2, 2, 1],
     padding="SAME")
edge_res = edge_image.numpy()[0, ..., 0]
plt.imshow(edge_res, cmap=plt.cm.gray)
```

Output will be as shown in the following *figure 4.5*. Notice that the contours or edges are quite hazy:

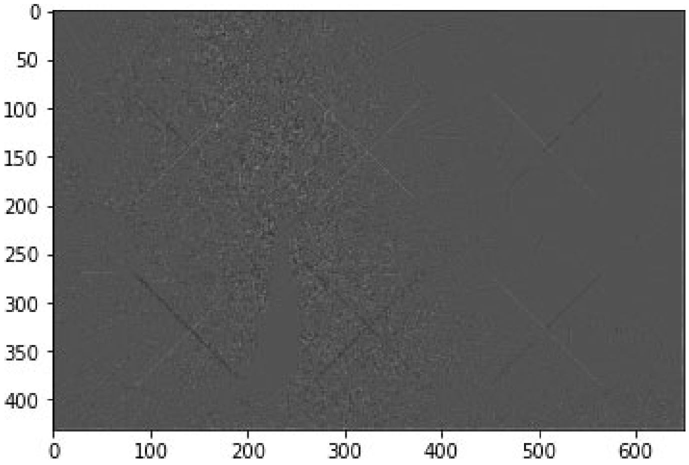


Figure 4.5: *Edge detection filter applied to the original image*

Let us look at another approach of down sampling, in the next section, to improve performance of the network.

Pooling

Convolutional layers apply learned filters to create feature maps. By stacking these layers, they learn low level features when they are closer to input layer, and more abstract features deeper into the model. Feature maps record precise location, which is quite sensitive to small movements in position of a feature. A common approach to address this problem is down sampling. Lower resolution of the image is created while retaining important elements. Pooling is a more robust approach to achieve the same result.

Down sampling can also be achieved by increasing the stride.

Stride is how far the filter moves at every step along one direction.

A pooling layer is generally added after one or more convolutional layers. A pooling layer operates on each feature map separately to create pooled feature maps.

It involves selecting a pooling operation or filter. Filter size is generally smaller than the feature map.

There are two types of pooling that are generally used:

- **Average pooling**: calculates the average value of each path in the feature map.

- **Max pooling**: calculates the maximum value for each patch in the feature map.

Using global pooling, instead of down sampling, reduces the pooling to a single value. In this case, the `pool_size` is sampled as the input feature map. Both `global.average` and `global.max` pooling are supported by TensorFlow 2.x with the classes `tf.keras.layers.GlobalAveragePooling2D` and `tf.keras.layers.GlobalMaxPooling2D`.

For max-pooling and average-pooling, the values in each window are aggregated into a single output, applying respectively the max or averaging operation. Once again, we use the low-level TensorFlow API to reproduce the results:

```
avg_pooled_image = tf.nn.avg_pool(image, ksize=[1, 2, 2, 1],
strides=[1, 2, 2, 1], padding="SAME")
avg_res = avg_pooled_image.numpy()[0, ..., 0]
plt.imshow(avg_res, cmap=plt.cm.gray)
```

The following figure shows the output after `average_pooling` has been applied to the image:

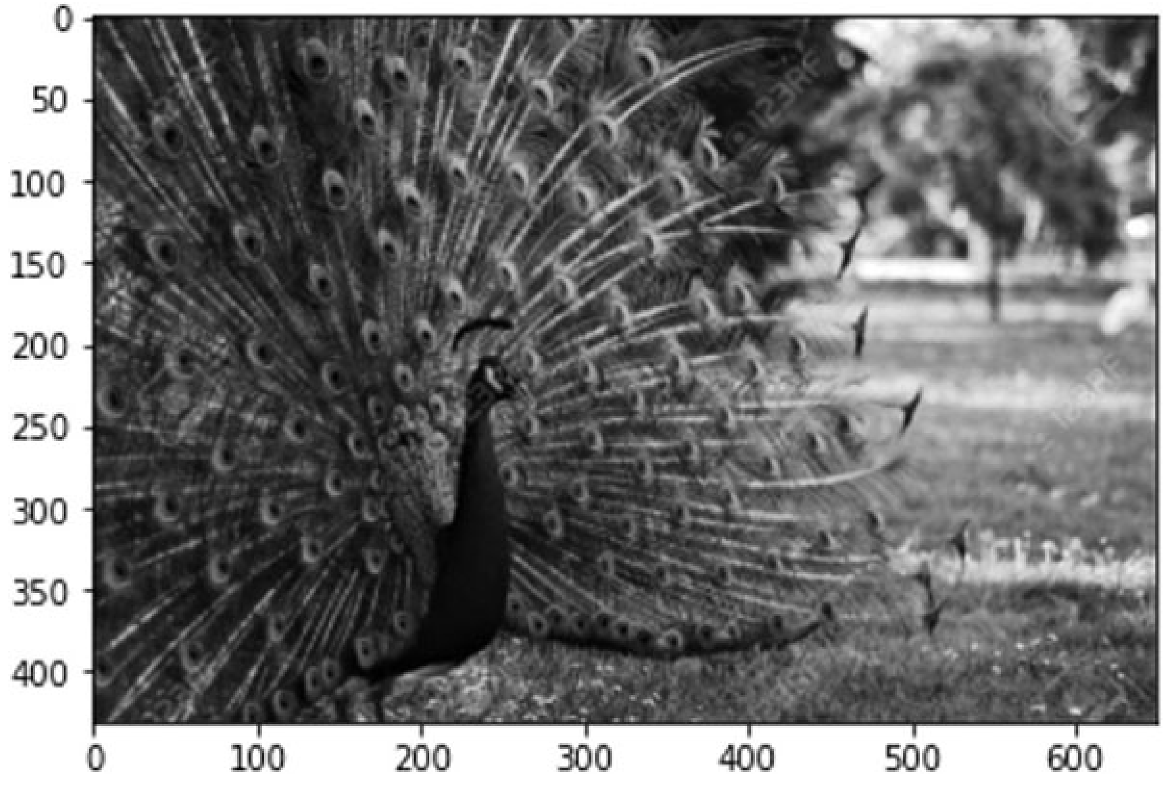


Figure 4.6: *Average pooling applied to the image*

With these hyper-parameters, the average pooling divided each dimension of the input image by *2*.

Let us perform max pooling next:

```
max_pooled_image = tf.nn.max_pool(image, ksize=[1, 10, 10, 1],
strides=[1, 2, 2, 1], padding="SAME")
max_res = max_pooled_image.numpy()[0, ..., 0]
plt.imshow(max_res, cmap=plt.cm.gray)
```

Output of the **`plt.imshow`** command will be as shown in the following *figure 4.7*:

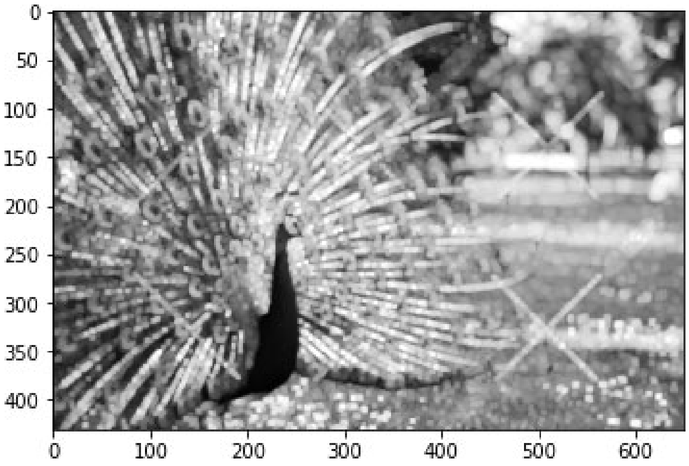


Figure 4.7: *Max pooling applied to the image*

As you can see, the picture is assigned the largest value for each window.

Simple convolutional network with TensorFlow

Let us start by building a simple CNN network with TensorFlow. Let us start with a classic example of computer vision—digit recognition, with the **Modified National Institute of Standards and Technology** (**MNIST**) dataset.

Preparing the data

Follow these steps:

1. First, we will import the data. It consists of *60,000 images* for the training set and *10,000 images* for the test set.

   It is easier to import TensorFlow with the alias **`tf`** for faster reading and typing. Here, **`x`** is used to denote the input data, and y to represent labels:

```
import tensorflow as tf
num_classes = 10
img_rows, img_cols = 28, 28
num_channels = 1
input_shape = (img_rows, img_cols, num_channels)
(x_train, y_train),(x_test, y_test) =
tf.keras.datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1))
```

2. The **`tf.keras.datasets`** module provides access to download and instantiate a number of classic datasets. After importing the data using **`load_data`**, notice that we divide the array by *255.0* to get a number in the range *[0, 1]* instead of *[0, 255]*. We will normalize data, either in the *[0, 1]* range or in the *[-1, 1]* range. If everything goes as expected, you will get an output similar to the following:

   ```
   Downloading data from
   https://storage.googleapis.com/tensorflow/tf-keras-
   datasets/mnist.npz
   11493376/11490434 [==============================] - 0s
   0us/step
   ```

Why normalization?

If we have two dimensions in our data and if one dimension *(a)* is much larger than the other dimension *(b)*, the gradient of the loss in terms of the weight of *a* will be larger than the gradient of the loss with respect to weight of *b*. The loss function is explained as follows:

A change in one weight, from the smaller dimension *b*, will lead to a small change in the loss. However, a small change in the other weight might lead to a very large change in the loss. This means gradient descent might take many small steps, with a very small learning rate, and zig-zag very slowly to the minimum.

3. We will run this sample in Google Colab. Let us make sure **`GPU`** is selected:

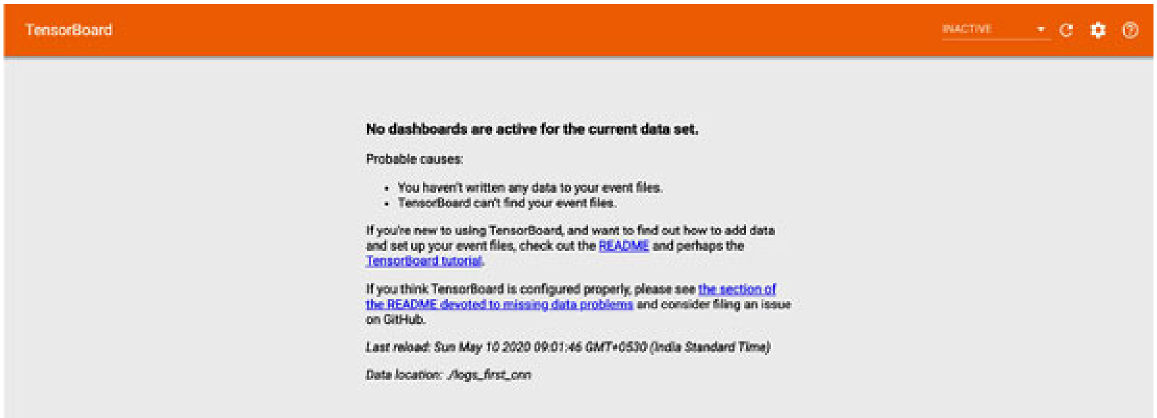
   ```
   device_name = tf.test.gpu_device_name()
   if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
   ```

```
print('Found GPU at: {}'.format(device_name))
```

4. Assuming you have GPU as the device configured, let us hook the TensorBoard:

```
import datetime, os
logs_base_dir = "./logs_first_cnn"
%load_ext tensorboard
os.makedirs(logs_base_dir, exist_ok=True)
%tensorboard --logdir {logs_base_dir}
```

The TensorBoard home page will show up, as displayed in the following *figure 4.8*:



No dashboards are active for the current data set.

Probable causes:

- You haven't written any data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the README and perhaps the TensorBoard tutorial.

If you think TensorBoard is configured properly, please see the section of the README devoted to missing data problems and consider filing an issue on GitHub.

Last reload: Sun May 10 2020 09:01:46 GMT+0530 (India Standard Time)

Data location: ./logs_first_cnn

***Figure 4.8:** TensorBoard loaded*

5. Let us create a model:

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(32, (3, 3),
activation='relu', kernel_initializer='he_uniform',
input_shape=(28, 28, 1)))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation='relu'))
model.add(tf.keras.layers.Dense(num_classes,
activation='softmax'))
```

Flattening a tensor means to remove all of the dimensions except for one.

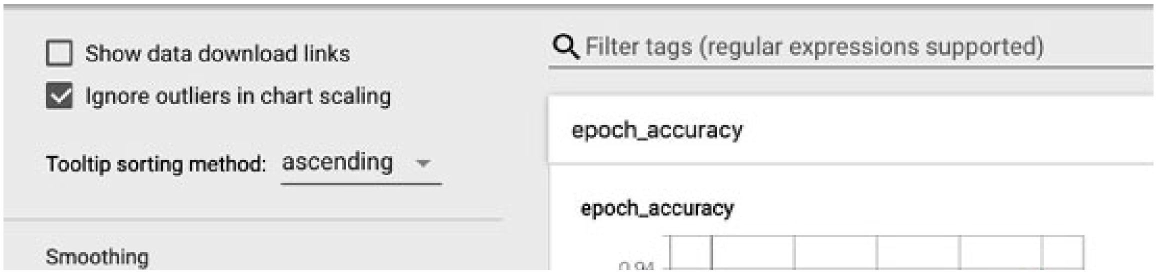
6. Compile it, and run it:

```
model.compile(optimizer='sgd',
  loss='sparse_categorical_crossentropy',
 metrics=['accuracy'])
logdir = os.path.join(logs_base_dir,
datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard_callback =
tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
model.fit(x_train, y_train, epochs=5, verbose=1,
validation_data=(x_test, y_test), callbacks=
[tensorboard_callback])
```

The output of the model training can be seen on the console as well as the attached TensorBoard:

```
Epoch 1/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.2821 - accuracy: 0.9170 - val_loss: 0.1587 -
val_accuracy: 0.9530
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.1453 - accuracy: 0.9580 - val_loss: 0.1169 -
val_accuracy: 0.9638
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.1089 - accuracy: 0.9684 - val_loss: 0.1015 -
val_accuracy: 0.9673
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.0881 - accuracy: 0.9747 - val_loss: 0.0846 -
val_accuracy: 0.9748
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step -
loss: 0.0740 - accuracy: 0.9784 - val_loss: 0.0761 -
val_accuracy: 0.9755
```

The TensorBoard shows the accuracy and loss over five epochs, as shown in the following *figure 4.9*:



☐ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: ascending

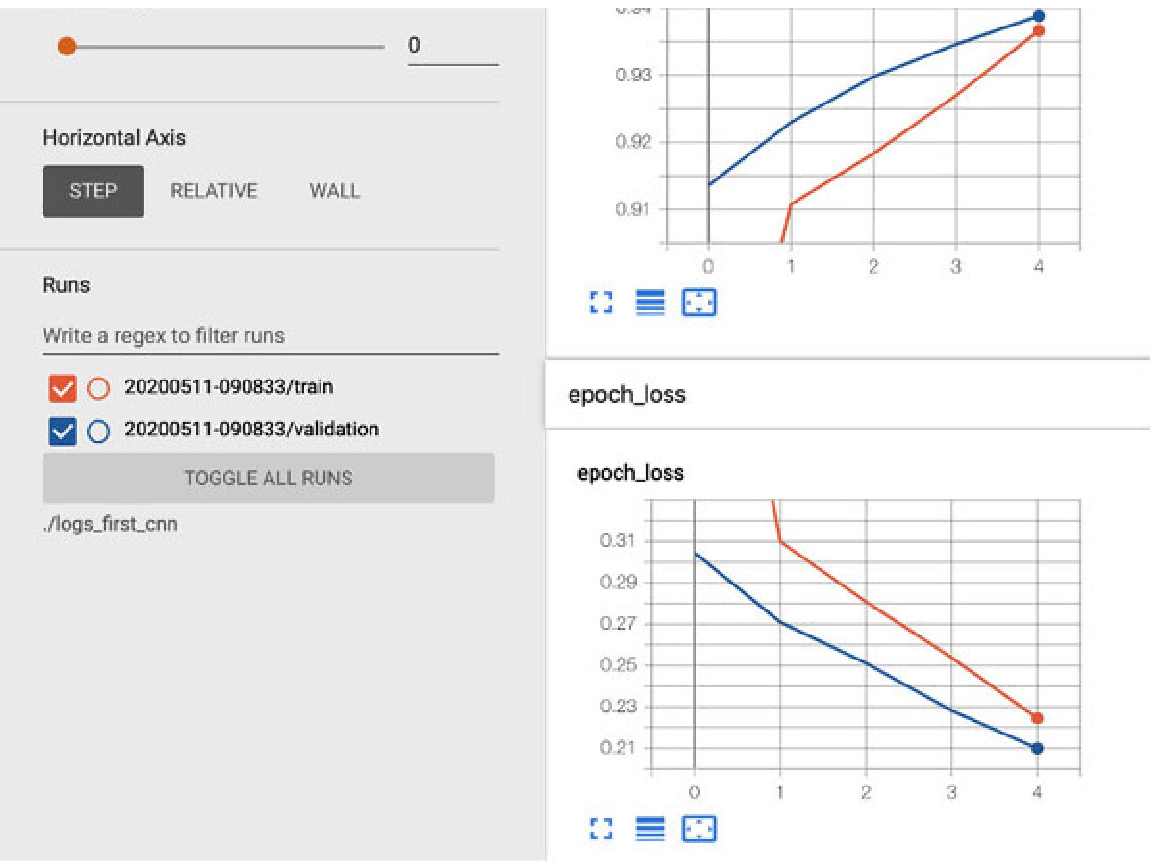


Figure 4.9: *Training and validation accuracy and loss for MNIST dataset*

Next, we will look at a more sophisticated CNN like ResNet.

Building ResNet with TensorFlow

Vanishing gradient was the problem with deeper networks. This was solved with ReLU activations and batch normalizations. ResNet has *152 layers*, but it still leads to a degradation problem which leads to higher classification errors.

Image degradation can be of various types. The examples of images after different image degradations are:

a. Gaussian white noise

b. Colored Gaussian noise

c. Salt and pepper noise

d. Motion blur

e. Gaussian blur

f. Degradation due to JPEG compression (JPEG quality)

It is a different problem as compared to overfitting because you see these issues in training accuracy graphs as well. Look at the following graphs:

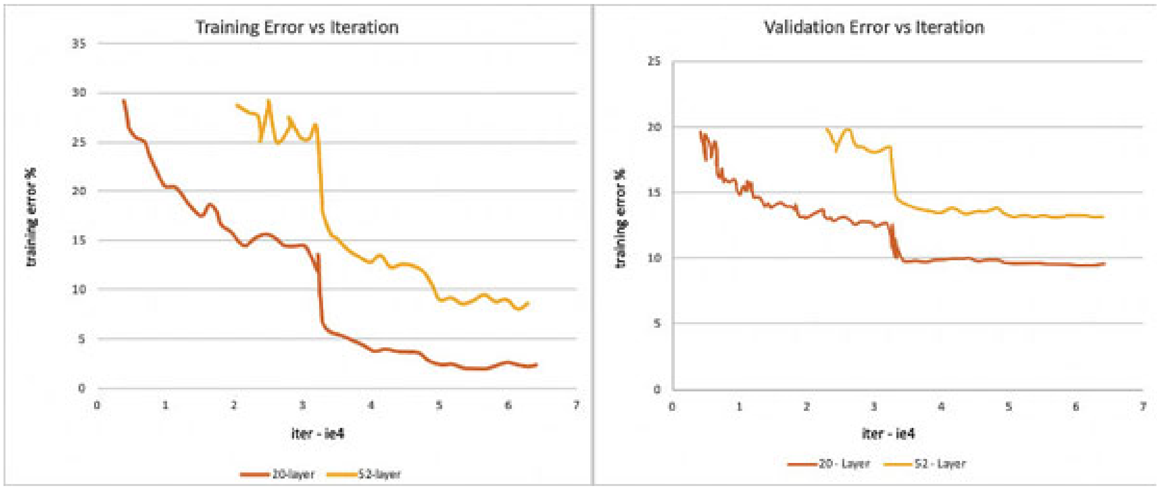


***Figure 4.10:** Graphs depicting overfitting due to adding more layers to a CNN network*

56-layer network has higher training and validation error. Neural networks are not very good at learning identity function, and beyond a few layers, the performance degrades as compared to shallower implementations. This is where **ResNet** comes into picture.

ResNet

ResNet makes the identity function explicit in the architecture. It consists of the following CNN blocks.

An identity function (or identity relation) is a function which returns the same value used as its argument. For *f* being an identity function:

$$f(x) = x$$

This equation holds true for all *x*.

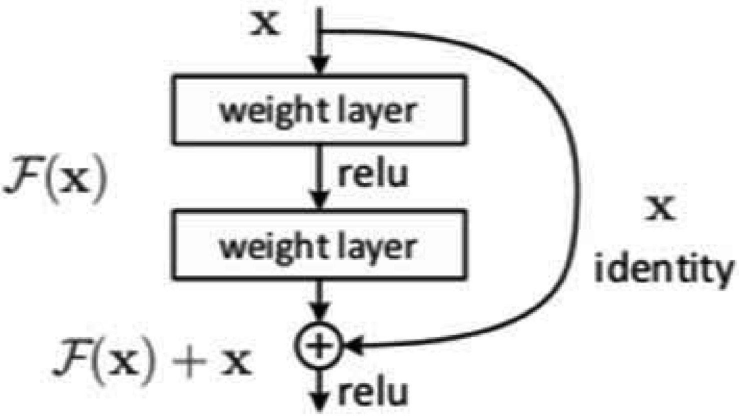

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 4.11: *Basic concept behind ResNet – adding identity matrix to the output of the convolution*

Let us implement ResNet for classifying *Cifar-10* dataset. We used Google Colab as we wanted to leverage GPUs in that environment. You can find the notebook at the following link:

https://colab.research.google.com/github/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/tensorflow2_resnet.ipynb#scrollTo=FJDZGKfH85Oj

1. Let us start by importing the packages needed from TensorFlow to build the model:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import datetime as dt
```

2. Test for the availability of the GPU device:

```
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

   If the GPU is available, you will get output like this: `Found GPU at: /device:GPU:0`.

3. Launch the TensorBoard with the following commands:

```
import datetime, os
LOG_DIR = './log'
logs_base_dir = "./logs"
%load_ext tensorboard
os.makedirs(logs_base_dir, exist_ok=True)
```

```
%tensorboard --logdir {logs_base_dir}
```

4. It will show up the next cell:

```
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.cifar10.load_data()
train_dataset = tf.data.Dataset.from_tensor_slices((x_train,
y_train)).batch(64).shuffle(10000)
train_dataset = train_dataset.map(lambda x, y: (tf.cast(x,
tf.float32) / 255.0, y))
train_dataset = train_dataset.map(lambda x, y:
(tf.image.central_crop(x, 0.75), y))
train_dataset = train_dataset.map(lambda x, y:
(tf.image.random_flip_left_right(x), y))
train_dataset = train_dataset.repeat()
```

5. Let us create a function to return the ResNet block. ResNet block is made of the following:

    - Conv2D layer which takes **`input_data`**
    - **`BatchNormalization`** on previous layer (helps reduce time taken to process the input data)
    - Another Conv2D layer with input from previous layer
    - Another **`BatchNormalization`** with input from previous layer
    - output of layer above and the **`input_data`**.
    - Apply ReLU activation on output of the previous layer:

    ```
    def res_net_block(input_data, filters, conv_size):
     x = layers.Conv2D(filters, conv_size,
     activation='relu',
      padding='same')(input_data)
     x = layers.BatchNormalization()(x)
     x = layers.Conv2D(filters, conv_size, activation=None,
     padding='same')(x)
     x = layers.BatchNormalization()(x)
     x = layers.Add()([x, input_data])
     x = layers.Activation('relu')(x)
     return x
    ```

6. Also, we will use the non res block to compare with the accuracy of ResNet based model. Notice the **`Add()`** layer is missing here:

```
def non_res_block(input_data, filters, conv_size):
```

```
x = layers.Conv2D(filters, conv_size, activation='relu',
 padding='same')(input_data)
x = layers.BatchNormalization()(x)
x = layers.Conv2D(filters, conv_size, activation='relu',
 padding='same')(x)
x = layers.BatchNormalization()(x)
return x
```

7. Create the model using combination of Conv2D layers and `10` ResNet blocks defined earlier:

```
Inputs = keras.Input(shape=(24, 24, 3))
x = layers.Conv2D(32, 3, activation=’relu’)(inputs)
x = layers.Conv2D(64, 3, activation=’relu’)(x)
x = layers.MaxPooling2D(3)(x)
num_res_net_blocks = 10
for I in range(num_res_net_blocks):
x = res_net_block(x, 64, 3)
x = layers.Conv2D(64, 3, activation’'rel’')(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256, activation’'rel’')(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(10, activation’'softma’')(x)
res_net_model = keras.Model(inputs, outputs)
```

8. Let us compile the model with callback tied to TensorBoard and call **`res_net_model.fit(..)`**:

```
res_net_model.compile(optimizer=keras.optimizers.Adam(),
    loss='sparse_categorical_crossentropy',
    metrics=['acc'])
logdir = os.path.join(logs_base_dir,
datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard_callback =
tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
res_net_model.fit(train_dataset, epochs=30,
steps_per_epoch=195,
    validation_data=valid_dataset,
    validation_steps=3, callbacks=[tensorboard_callback])
```

You will see the output in TensorBoard with the text log showing up in Colab:

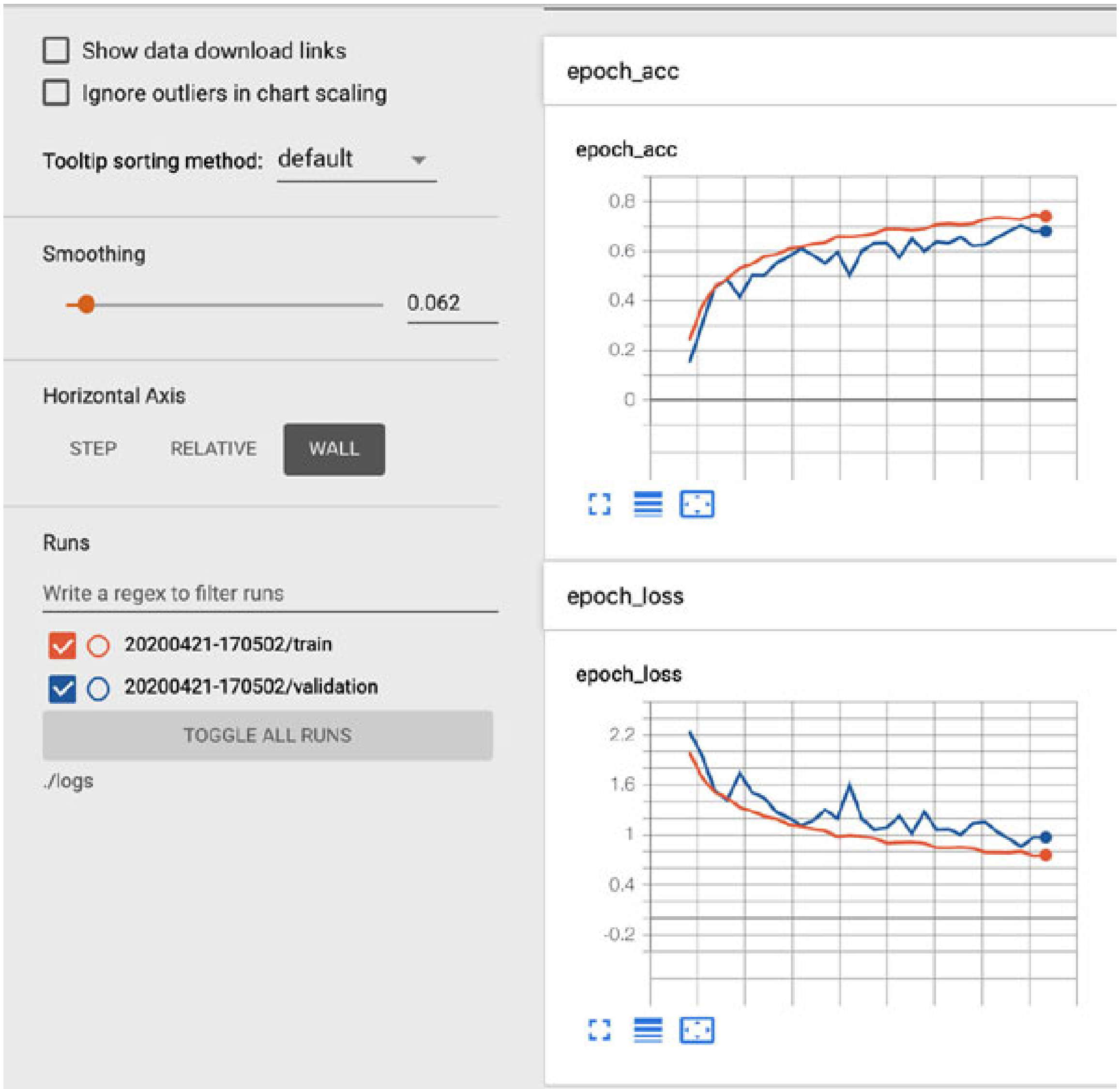
```
Epoch 28/30
```

```
195/195 [==============================] - 3s 14ms/step -
loss: 0.7910 - acc: 0.7304 - val_loss: 0.8671 - val_acc:
0.6981
Epoch 29/30
195/195 [==============================] - 3s 15ms/step -
loss: 0.7533 - acc: 0.7423 - val_loss: 0.9251 - val_acc:
0.6737
Epoch 30/30
195/195 [==============================] - 3s 14ms/step -
loss: 0.7508 - acc: 0.7433 - val_loss: 0.9072 - val_acc:
0.6941
<tensorflow.python.keras.callbacks.History at
0x7f09de141b00>
```

9. Next, let us run the same data on **`non_res_net`**:

```
inputs = keras.Input(shape=(24, 24, 3))
x = layers.Conv2D(32, 3, activation='relu')(inputs)
x = layers.Conv2D(64, 3, activation='relu')(x)
x = layers.MaxPooling2D(3)(x)
num_res_net_blocks = 10
for i in range(num_res_net_blocks):
     x = non_res_block(x, 64, 3)
x = layers.Conv2D(64, 3, activation='relu')(x)
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dense(256, activation='relu')(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(10, activation='softmax')(x)
non_res_model = keras.Model(inputs, outputs)
callbacks = [
     # Write TensorBoard logs to `./logs` directory
 keras.callbacks.TensorBoard(log_dir='./log/{}'.format(dt.da
 tetime.now().strftime("%Y-%m-%d-%H-%M-%S")),
 write_images=True),
]
```

The following *figure 4.12* shows the output plotted on TensorBoard:



□ Show data download links

□ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.062

Horizontal Axis: STEP RELATIVE [✓ WALL]

Runs

Write a regex to filter runs

☑ ○ 20200421-170502/train

☑ ○ 20200421-170502/validation

TOGGLE ALL RUNS

./logs

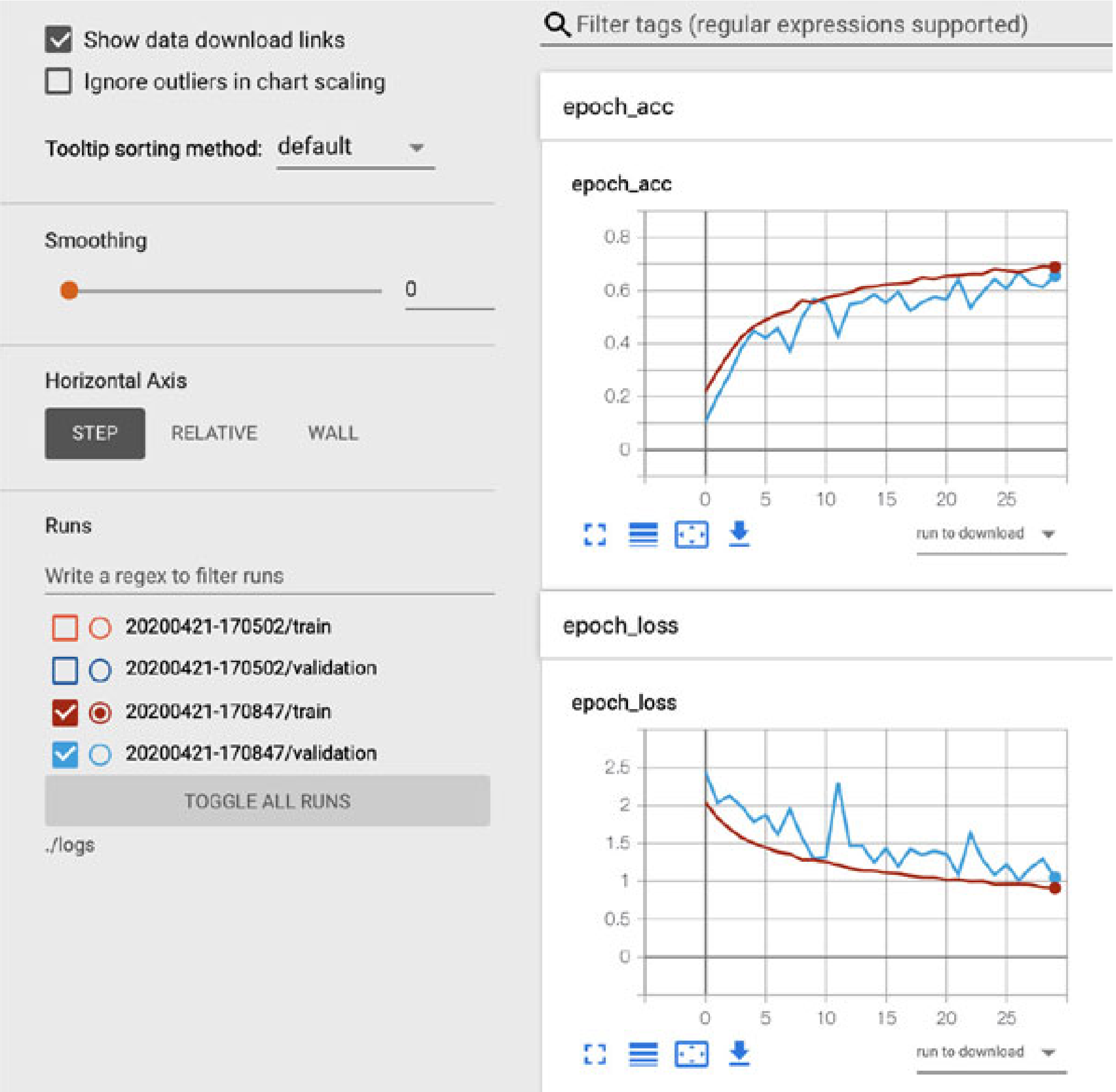
Figure 4.12: *Training and validation accuracy and loss for ResNet based model*

```
non_res_model.compile(optimizer=keras.optimizers.Adam(),
 loss='sparse_categorical_crossentropy',
 metrics=['acc'])
non_res_model.fit(train_dataset, epochs=30,
steps_per_epoch=195,
     validation_data=valid_dataset,
     validation_steps=3, callbacks=[tensorboard_callback])
```

The output will show up on the Colab cell as well as TensorBoard, as shown in the following listing and *figure 4.13*:

```
Epoch 28/30
```

```
195/195 [==============================] - 4s 19ms/step -
loss: 0.9399 - acc: 0.6823 - val_loss: 1.0587 - val_acc:
0.6485
Epoch 29/30
195/195 [==============================] - 4s 20ms/step -
loss: 0.9440 - acc: 0.6756 - val_loss: 1.0689 - val_acc:
0.6533
Epoch 30/30
195/195 [==============================] - 4s 19ms/step -
loss: 0.8970 - acc: 0.6976 - val_loss: 1.0955 - val_acc:
0.6397
```



☑ Show data download links

□ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing: 0

Horizontal Axis: [✓ STEP] RELATIVE WALL

Runs

□ ○ 20200421-170502/train

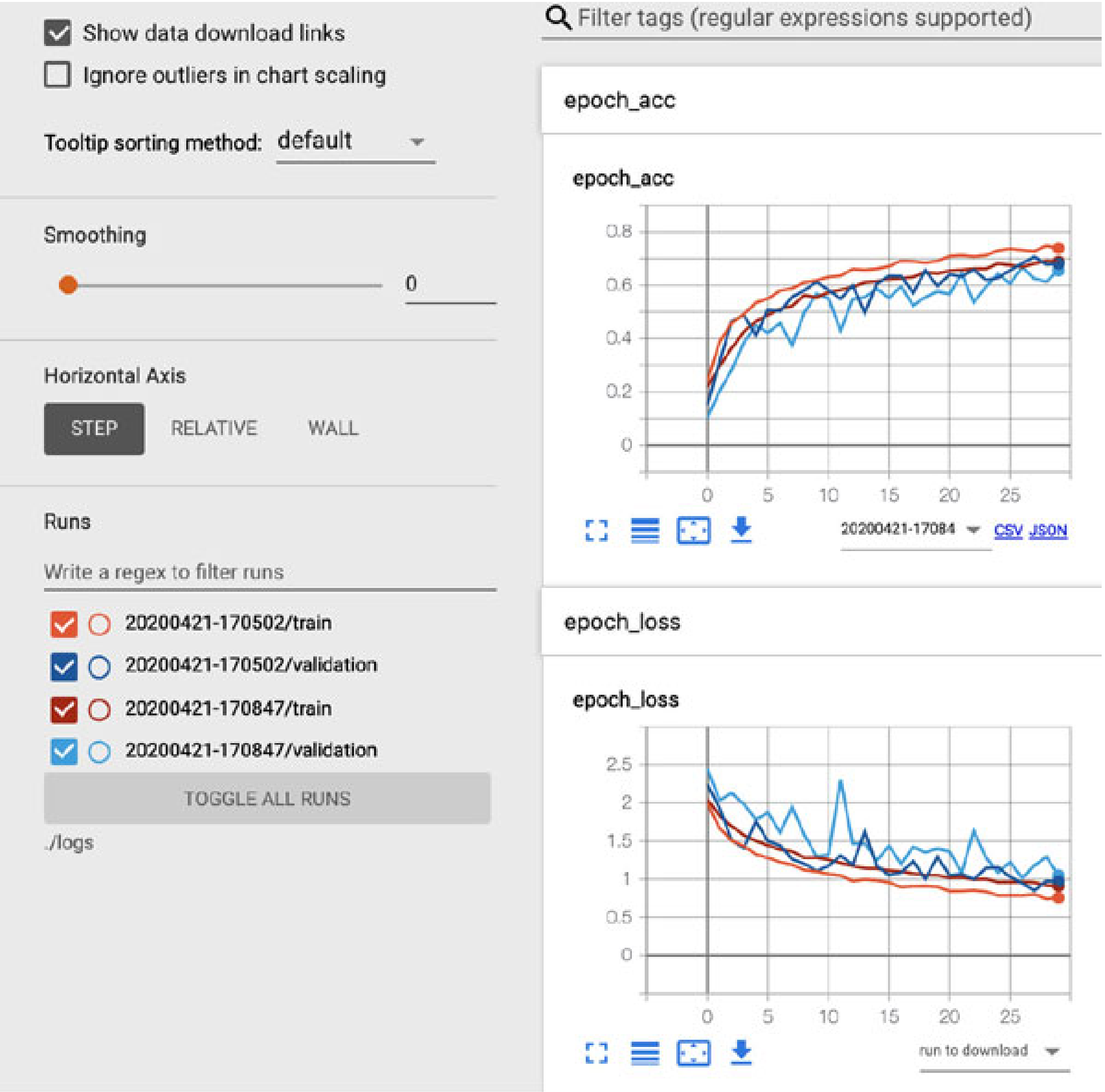
□ ○ 20200421-170502/validation

☑ ● 20200421-170847/train

☑ ○ 20200421-170847/validation

Figure 4.13: *Training and validation accuracy and loss for not_res_net model*

The accuracy of ResNet is higher than the non ResNet model, as seen in the preceding numbers. Let us compare the ResNet and non ResNet plots:



☑ Show data download links

☐ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0

Horizontal Axis: STEP RELATIVE WALL

Runs

Write a regex to filter runs

☑ ○ 20200421-170502/train

☑ ○ 20200421-170502/validation

☑ ○ 20200421-170847/train

☑ ○ 20200421-170847/validation

./logs

Figure 4.14: *ResNet and non ResNet plots*

In the next section, we will talk about some of the other advanced models for computer vision.

Advanced computer vision techniques

Research in computer vision has been moving forward both through incremental contributions and large innovative leaps. Challenges organized by researchers and companies, inviting experts to submit new solutions in order to best solve a predefined task, have been playing a key role in triggering such instrumental contributions. The **ImageNet classification** contest [**ImageNet Large Scale Visual Recognition Challenge** (**ILSVRC**)] is a perfect example. With its millions of images split into 1000 fine-grained classes, it still represents a great challenge for daring researchers, even after the significant and symbolic victory of *AlexNet* in *2012*.

In this section, we will discuss some of the classic deep learning methods that followed AlexNet in tackling ILSVRC, covering the reasons leading to their development, and the contributions they made.

VGG architecture

VGG16 is a convolutional neural network model proposed by *K. Simonyan* and *A. Zisserman* from the *University of Oxford* in the paper *Very Deep Convolutional Networks for Large-Scale Image Recognition [1]*. The model achieves *92.7%* top five test accuracy in ImageNet, which is a dataset of over *14 million* images belonging to *1000 classes*. It was one of the most famous models submitted to *ILSVRC-2014* *(***http://www.image-net.org/challenges/LSVRC/2014/results***)*. It makes an improvement over AlexNet by replacing large kernel-sized filters (*11* and *5* in the first and second convolutional layer, respectively) with multiple *3× 3* kernel-sized filters one after another. VGG16 was trained for weeks and was using *NVIDIA Titan Black GPU's*.

Motivation behind VGG

AlexNet is the name of a CNN designed by *Alex Krizhevsky*, and published with *Ilya Sutskever* and *Geoffrey Hinton*.

AlexNet competed in the *ImageNet Large Scale Visual Recognition Challenge* (**https://en.wikipedia.org/wiki/ImageNet_Large_Scale_Visual_Recognition_Challenge**) in *September 2012*. The network was able to achieve a top five error of *15.3%*, this was *10.8* percentage points lower than the runner up which had it as *26.1%*. The original paper's primary conclusion was that the depth in terms of number of layers of the model was required for its high performance. This was computationally expensive, but made feasible due to the utilization of **graphics processing units** (**GPUs**) during the model's training.

Some of the innovations introduced by AlexNet were as follows:

- the use of a **rectified linear unit** (**ReLU**) as an activation function, which prevents the vanishing gradient problem (explained later in this chapter), and thus improving training (compared to using `sigmoid` or `tanh`)
- the application of dropout to CNNs.
- the typical CNN architecture combining blocks of convolution and pooling layers with dense layers afterwards for the final prediction
- the application of random transformations (image translation, horizontal flipping, and more) to synthetically augment the dataset (that is, augmenting the number of different training images by randomly editing the original samples)

It was clear that this prototype architecture had room for improvement. The main motivation of many researchers was to try to go deeper (that is, building a network composed of a larger number of stacked layers), despite the challenges arising from this. Indeed, more layers typically means more parameters to train, making the learning process more complex. However, *Karen Simonyan* and *Andrew Zisserman* from *Oxford*'s VGG group tackled this challenge with success. The method they submitted to *ILSVRC 2014* reached the top five errors of *7.3%,* dividing the *16.4%* error of AlexNet by more than *50%*.

The top five accuracy is one of the main classification metrics of ILSVRC. It considers that a method has predicted properly if the correct class is among its five first guesses. Indeed, for many applications, it is fine to have a method that's able to reduce a large number of class candidates to a lower number (for instance, to leave the final choice between the remaining candidates to an expert user). The top five metrics are a specific case of the more generic *top-k metrics*.

Simonyan and *Zisserman,* in their paper *Very Deep Convolutional Networks for Large-Scale Image Recognition, ArXiv, 2014 [1],* presented how they developed their network to be deeper than most previous ones. They introduced six different CNN architectures, from *11* to *25* layers deep. Each network is composed of five blocks of several consecutive convolutions followed by a max-pooling layer and three final dense layers (with dropout for training).

All the convolutional and max-pooling layers have **SAME** for padding. The convolutions have *s = 1* for stride, and are using the ReLU function for activation.

SAME and VALID padding are some of the most common types of padding used in Convolution Neural Network (CNN) models. These padding types are mainly used in TensorFlow.

The general meaning of SAME and VALID padding are as follows:

- **SAME: The padding is such that the input and the output is of the same size (provided *stride=1*) (*pad value = kernel size*).**
- **VALID: The padding value is set to 0.**

SAME padding implementation is different depending on the framework being considered.

VGG network is represented in the *figure 4.16*: **VGG16 architecture**.

The two most performant architectures, still commonly used nowadays, are called **VGG16** and **VGG19**. The numbers (**16** and **19**) represent the depth of these CNN architectures; that is, the number of trainable layers stacked together as shown in the following *figure 4.15*:

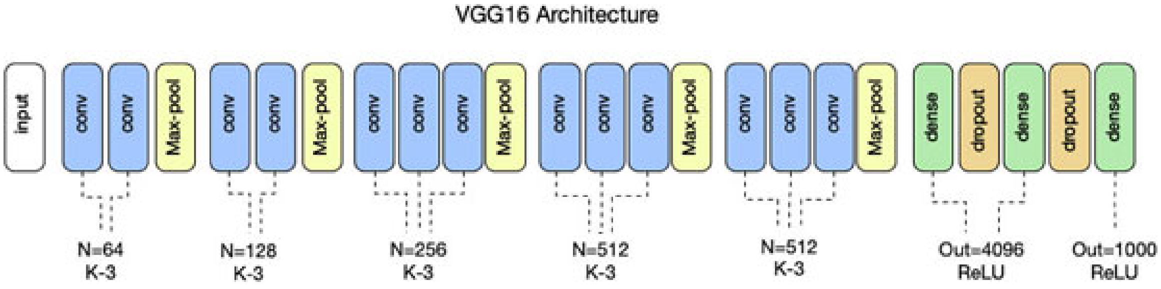


Figure 4.15: *VGG16 architecture*

VGG16 has approximately *138 million* parameters and VGG19 have *144 million* parameters. These numbers of parameters are quite high, although, we will explore how the VGG researchers took a new approach to keep these values in check, despite the depth of their architecture.

Learning from the VGG network

Authors observed that a convolution with a stack of two *3x3* kernels have the same effective receptive field as a convolution with single *5x5* kernel. Therefore, VGG network had more but smaller convolutions. This led to:

- decrease in parameters.
- increased nonlinearity. (Larger number of convolutional layers with nonlinear activation function led the network to learn more complex

features.)

Increased depth of feature maps

VGG authors doubled the depth of feature maps for each block from *64 convolution* to *512 convolutions*. Max-pooling was used to reduce the spatial dimensions. This allowed encoding of spatial information into more and more complex and discriminate features.

VGG in TensorFlow

TensorFlow supports both VGG16 and VGG19. We will be using the following sample dataset to show how to use these pre-built models.

We will use a transfer learning to classify medical imaging data PCam to detect cancer.

Dataset

We will use the data for this **PatchCamelyon** (**PCam**) benchmark dataset (**https://github.com/basveeling/pcam**).

PCam is highly interesting for its size, simplicity to get started on, and approachability.

Why PCam?

Fundamental machine learning advancements are predominantly evaluated on straight-forward natural-image classification datasets. Medical imaging is becoming one of the major applications of ML, and we believe it deserves a spot on the list of **go-to** ML datasets both to challenge future work and to steer developments into directions that are beneficial for this domain.

Approach

We will be using pre-built model weights (from various architectures like VGG16, inception V3), translate the models to work with new dataset, and add additional tuning in the form of dropout rate, loss function to come up with the optimized model with an accuracy of *79%* or above.

Dataset shape - 360 images with binary labels

We will load 360 images and prepare them for consumption by various models:

1. First, we need to extract the ID of the image from the path. We have prepared the notebook to run both locally as well as on Google Colab:

```
if IN_COLAB:
 BASE = '/content/gdrive/My
 Drive/cancer_detection/metastatic_cancer'
else:
 BASE = '.'
```

Set the directories for storing various outputs:

```
import tensorflow
tensorflow.__version__
VERSION = tensorflow.__version__
# Output files
model_type='vgg16'
no_of_images = 'all'
EPOCHS = 25
if IN_COLAB:
 PLOTS = 'plots_2.6.0_google_collab'
else:
 PLOTS = 'plots_2.6.0'
_APPEND = '_' + model_type + '_' + str(no_of_images) + '_' +
str(EPOCHS)
APPEND = _APPEND + ".png"
if IN_COLAB:
 if not os.path.exists(BASE + "/training_logs_" + VERSION):
  os.mkdir(BASE + "/training_logs_" + VERSION)
 if not os.path.exists(BASE + "/model_summary/"):
  os.mkdir(BASE + "/model_summary/")
 if not os.path.exists(BASE + "/model_summary/" +
 "model_summary_" + VERSION):
  os.mkdir(BASE + "/model_summary/" + "model_summary_" +
  VERSION)
 if not os.path.exists(BASE + '/' + PLOTS):
  os.mkdir(BASE + '/' + PLOTS)
if IN_COLAB:
 TRAINING_LOGS_FILE = BASE + "/training_logs_" + VERSION +
 '/training_logs' + _APPEND + '.csv'
 MODEL_SUMMARY_FILE = BASE + "/model_summary/"
 "model_summary_" + VERSION + "/model_summary" + _APPEND +
 ".txt"
 MODEL_PLOT_FILE = BASE + '/' + PLOTS + "/model_plot_" +
 APPEND
 MODEL_FILE = "model_" + VERSION +
 "/model_vgg16_all_collab.h5"
 TRAINING_PLOT_FILE = BASE + '/'+ PLOTS + "/training" +
 APPEND
 VALIDATION_PLOT_FILE = BASE + '/'+ PLOTS + "/validation" +
 APPEND
 ROC_PLOT_FILE = BASE + '/'+ PLOTS + "/roc" + APPEND
else:
 TRAINING_LOGS_FILE = "training_logs_" + VERSION +
 '/training_logs' + _APPEND + '.csv'
 MODEL_SUMMARY_FILE = "model_summary_" + VERSION +
 "/model_summary" + _APPEND + ".txt"
 MODEL_PLOT_FILE = PLOTS + "/model_plot_" + APPEND
 MODEL_FILE = "model_" + VERSION +
 "/model_vgg16_all_collab.h5"
 TRAINING_PLOT_FILE = PLOTS + "/training" + APPEND
 VALIDATION_PLOT_FILE = PLOTS + "/validation" + APPEND
 ROC_PLOT_FILE = PLOTS + "/roc" + APPEND
data_dir_training = pathlib.Path(BASE + '/training')
```

2. Let us plot the images from two classes, `0` and `1`:

```
import PIL
import PIL.Image
zeros = list(data_dir_training.glob('0/*'))
PIL.llImage.open(str(zeros[0]))
```

The following *figure 4.16* shows the first image from class `0`:

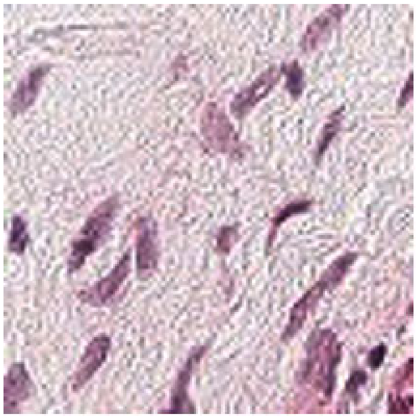


__Figure 4.16:__ First image from class 0

Next, we look at class **1**:

```
one = list(data_dir_training.glob('1/*'))
PIL.Image.open(str(one[0]))
```

Output will be like the following *figure 4.17*. You will notice even visually that the image looks quite different:

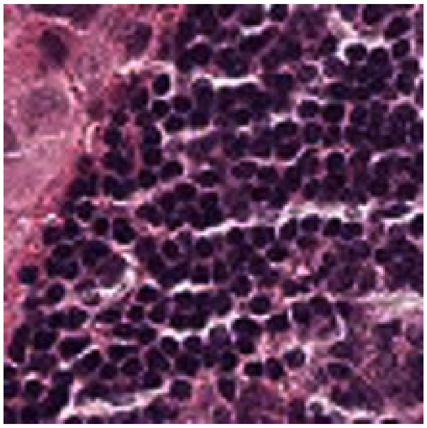


__Figure 4.17:__ First image from class 1

3. We will use **ImageDataGenerator** to create random variations of the dataset for better accuracy:

```
training_generator =
training_data_generator.flow_from_directory(
        training_path,
        target_size=(IMAGE_SIZE2,IMAGE_SIZE2),
        batch_size=BATCH_SIZE,
        lass_mode='binary')
```

The **ImageDataGenerator** class in Tensorflow Keras is used for implementing image augmentation. The major advantage is its ability to produce real-time image augmentation. This means it can generate augmented images dynamically during the training of the model making the overall mode more robust and accurate.

If this class was not present, user would have to manually generate the augmented image as a pre-processing step, and include them in our training dataset. But in TensorFlow, ImageDataGenerator takes care of this automatically during the training phase. And it does all this with efficient memory management so that the user can train a huge dataset efficiently with lesser memory consumption.

There are five techniques for applying image augmentation:

- Random rotation
- Random shift
- Vertical shift
- Random flips
- Random brightness
- Random zoom

Use this to generate data for validation and testing

We will use this technique to generate training, validation, and testing data:

```
# Data generation
training_generator = training_data_generator.flow_from_directory(
       training_path,
        target_size=(IMAGE_SIZE2,IMAGE_SIZE2),
        batch_size=BATCH_SIZE,
        class_mode='binary')
validation_generator =
ImageDataGenerator(rescale=1./255).flow_from_directory(
       validation_path,
        target_size=(IMAGE_SIZE2,IMAGE_SIZE2),
        batch_size=BATCH_SIZE,
        class_mode='binary')
testing_generator =
ImageDataGenerator(rescale=1./255).flow_from_directory(
       validation_path,
       training_path,
        target_size=(IMAGE_SIZE2,IMAGE_SIZE2),
        batch_size=BATCH_SIZE,
        class_mode='binary')
        shuffle=False)
```

The preceding datasets generated will be used for **VGG16**, **VGG19**, and **Inception_v3**.

VGG16

The topology of the convolution network using VGG16, which we will use in our experiment, is shown in the following *figure 4.18*:

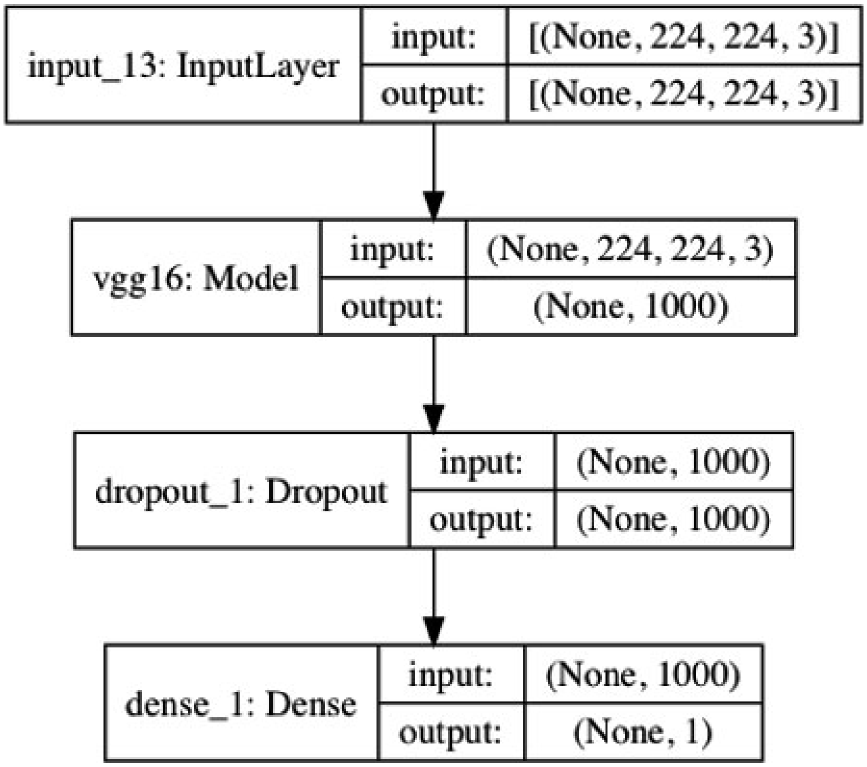


Figure 4.18: *VGG16 based model topology*

Details of each layer are described as follows:

- Input layer will be rescaled images from *96x96x3* to *224x224x3,* which is the standard input shape by VGG.
- The output of the VGG16 model will be (**`None`**, **`1000`**), where **`None`** will be replaced by the batch size being processed.
- We will apply a dropout of *0.5*.
- The last layer will be the dense layer with the output shape of (**`None`**, **`1`**) – **`1`** being either **`0`** or **`1`** binary classification output label:

```
# Model
input_shape = (IMAGE_SIZE2, IMAGE_SIZE2, 3)
inputs = Input(input_shape)
vgg16 = VGG16(include_top=False, input_shape=(224, 224, 3))
(inputs)
outputs = GlobalAveragePooling2D()(vgg16)
outputs = Dropout(0.5)(outputs)
outputs = Dense(1, activation='sigmoid')(outputs)
model = Model(inputs, outputs)
model.compile(optimizer=Adam(lr=0.0001, decay=0.00001),
      loss='binary_crossentropy',
      metrics=['accuracy'])
model.summary()
plot_model(model,
      to_file=MODEL_PLOT_FILE,
      show_shapes=True,
      show_layer_names=True)
```

Other parameters used are listed as follows. They help define the batch size for training, verbosity of the output, total sample count, and testing the sample count:

```
# Hyperparams
IMAGE_SIZE2 = 224
BATCH_SIZE = 192
VERBOSITY = 1
TESTING_BATCH_SIZE = 50
```

The output of the program once run will show training and validation loss and accuracy:

```
Epoch 00024: val_accuracy improved from 0.72500 to 0.75000,
saving model to model_2.6.0/model_vgg16_all_collab.h5
Epoch 25/25
2/2 [==============================] - 6s 2s/step - loss:
0.4610 - accuracy: 0.8278 - val_loss: 0.6795 - val_accuracy:
0.6750
Epoch 00025: val_accuracy did not improve from 0.75000
```

Next, we will plot the graph for training and validation accuracy:

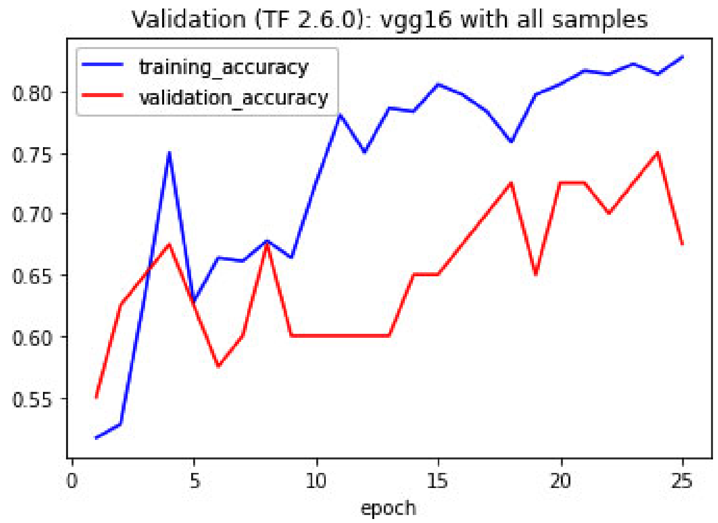


Figure 4.19: *Test and validation accuracy for VGG16 based model*

The training and validation accuracy are **0.8278** and **0.7500** (epoch 24). Next, we will look at the ROC curve.

Area under the curve for VGG16

AUC for this model is coming to around **0.807** as shown in the *figure 4.20*, which is considered quite decent:

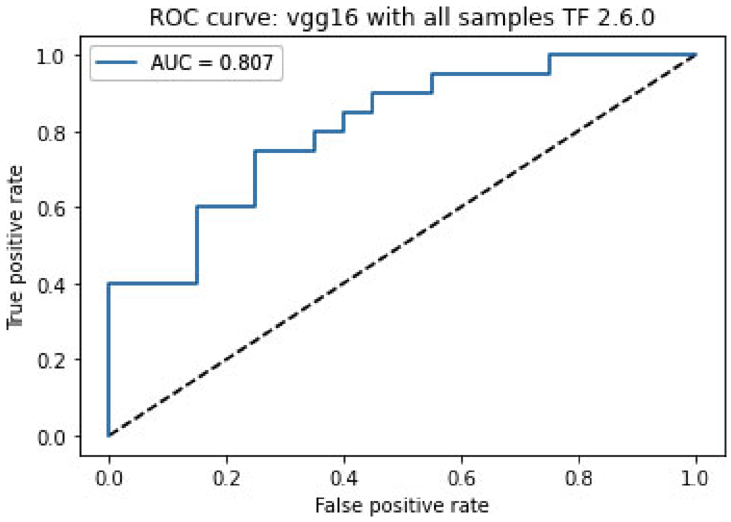


Figure 4.20: *ROC curve for VGG16 based model*

VGG16 is able to give an accuracy of about *74%* but is a very expensive network to load and train on, from memory and speed perspective.

Let us look at VGG19 and inception V3 as an alternative for transfer learning.

VGG19

VGG19 is a convolutional neural network trained on ImageNet dataset. It has *19 layers* and can classify images into up to *1000 categories*. This network is slightly newer than VGG16 but has three additional layers. Let us plot the training and validation accuracy:

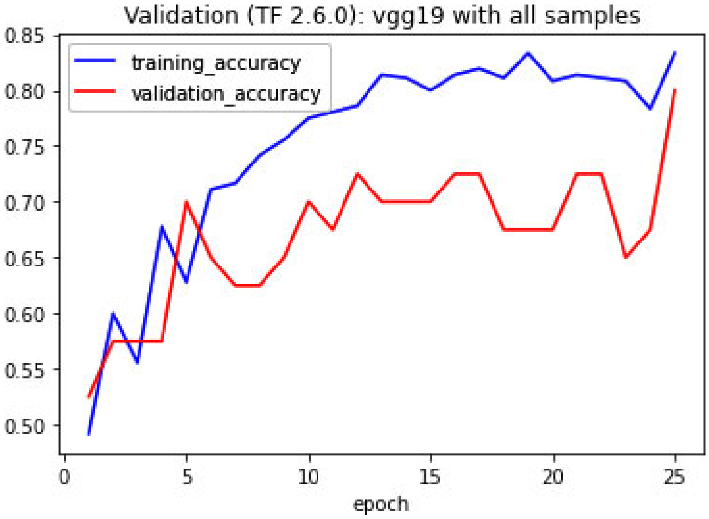


Figure 4.21: *Test and validation accuracy for VGG19 based model*

Final metrics are listed as follows.

Validation accuracy is worse than VGG16:

- `loss`: `0.4062`
- `accuracy`: `0.8333`
- `val_loss`: `0.9118`
- `val_accuracy`: `0.8000`

Let us plot the ROC curve now as shown in the *figure 4.22*:

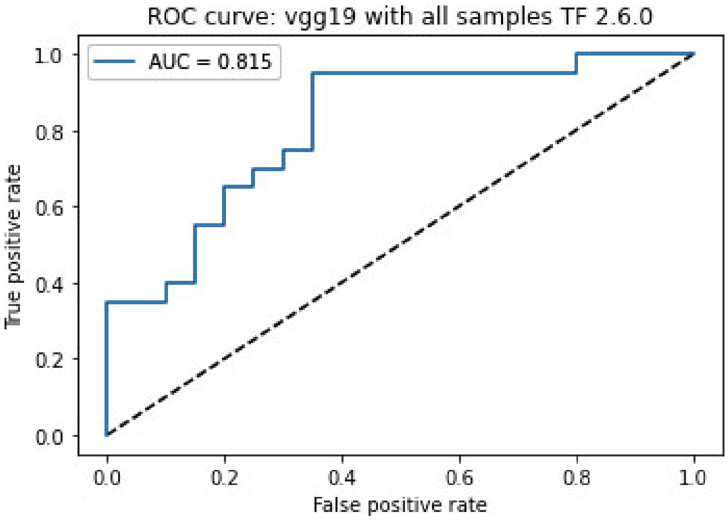


Figure 4.22: *ROC curve for VGG19 based model*

Area under the curve for VGG19 is higher than VGG16, though it takes more time to train and will take more space (memory). This means, for the ImageNet sample, deeper network helped in *TF 2.6.0*.

Inception V3

The *inception* micro-architecture was introduced by the team at *Google*, *Szegedy et al.* in the *2014* paper, *Going Deeper with Convolutions*.

The inception module acts as a multi-level feature extractor by using *1× 1*, *3× 3*, and *5× 5* convolutions in the same module of the network. The output of these filters is stacked along the channel dimension before being fed into the next layer in the network.

The original manifestation of this architecture was called **GoogleNet**. Subsequent manifestations have been called **Inception vN**, where *N* refers to the version number put out by *Google*.

The *Inception V3* architecture is from the new publication by *Szegedy et al.* Rethinking the *Inception Architecture for Computer Vision (2015) (***https://arxiv.org/abs/1512.00567***)*, it proposed updates to the inception module to further boost the ImageNet classification accuracy.

The weights for InceptionV3 are smaller than other networks like VGG and ResNet, coming in at 96 MB.

InceptionV3 architecture

We are planning to load the model from a pre-trained file that has an input dimension of (*None, 96, 96, 1*). Let us look at the following implementation. InceptionV3 is a part of **`tf.keras.applications`**:

```
# Model
input_shape = (IMAGE_SIZE2, IMAGE_SIZE2, 3)
inputs = Input(input_shape)
inceptionv3 = InceptionV3(include_top=False, input_shape=(224,
224, 3))(inputs)
outputs = GlobalAveragePooling2D()(inceptionv3)
outputs = Dropout(0.5)(outputs)
outputs = Dense(1, activation='sigmoid')(outputs)
model = Model(inputs, outputs)
model.compile(optimizer=Adam(lr=0.0001, decay=0.00001),
     loss='binary_crossentropy',
     metrics=['accuracy'])
model.summary()
plot_model(model,
   to_file=MODEL_PLOT_FILE,
   show_shapes=True,
     show_layer_names=True)
```

The **Inception_v3** model has the output of (**`None`**, **`1`**, **`1`**, **`2048`**), on which global average pooling is applied.

The purpose of global average pooling is different than that of ordinary pooling. The goal of this technique is to not make a network invariant to small translations or perturbations of an image. It also doesn't make a relatively small reduction in the size of an array of pixels, as the common *2×2* max-pooling mechanism does.

The following *figure 4.23* shows the summary on input and input layer for each layer:

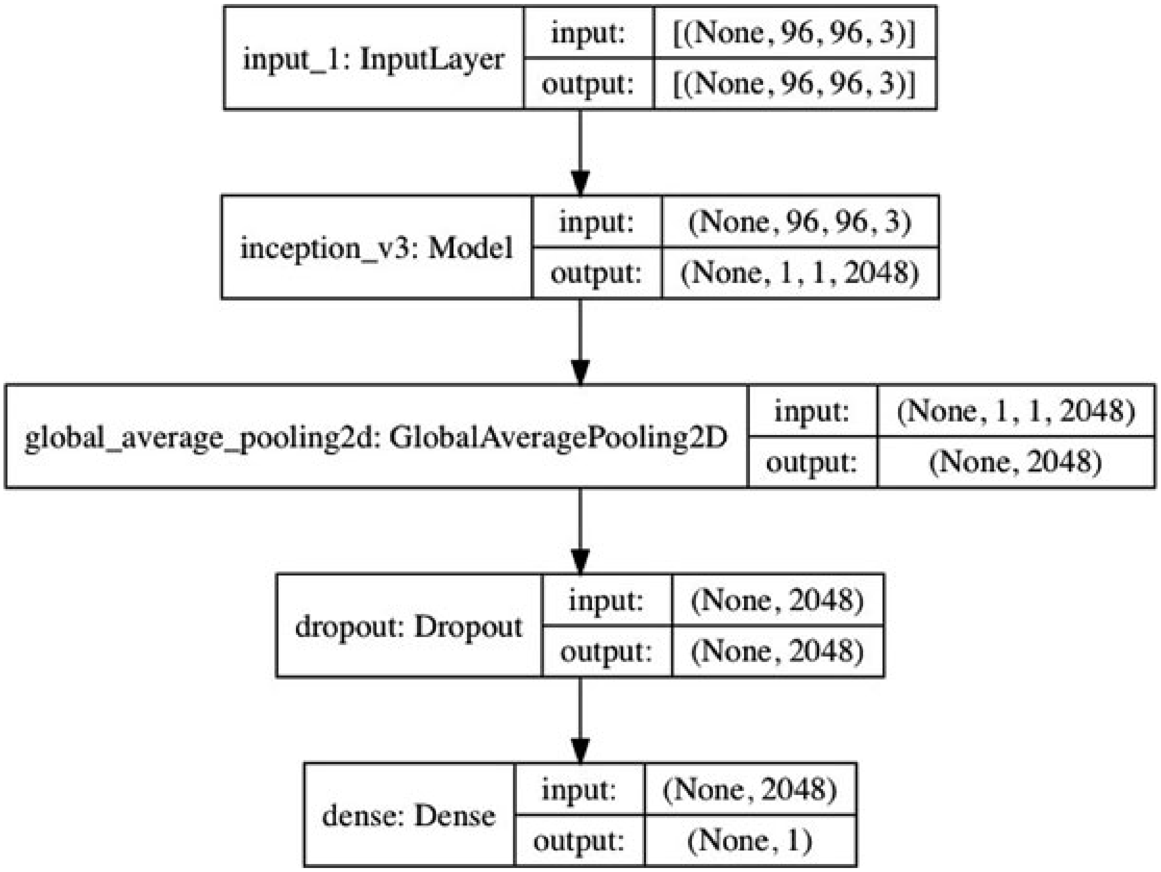


Figure 4.23: *Plot of model created of model InceptionV3*

Let us fit the model:

```
history = model.fit(training_generator,
      steps_per_epoch=len(training_generator),
      validation_data=validation_generator,
      validation_steps=len(validation_generator),
       epochs=EPOCHS,
       verbose=VERBOSITY,
       callbacks=[#PlotLossesKeras(),
        ModelCheckpoint(MODEL_FILE,
         monitor='val_accuracy',
         verbose=VERBOSITY,
         save_best_only=True,
         mode='max'),
        CSVLogger(TRAINING_LOGS_FILE,
         append=True,
         separator=';')
        ])
```

Output of the training is as follows:

```
Epoch 00023: val_accuracy did not improve from 0.87500
Epoch 24/25
2/2 [==============================] - 20s 9s/step - loss: 0.2424
- accuracy: 0.9056 - val_loss: 0.4493 - val_accuracy: 0.8750
Epoch 00024: val_accuracy did not improve from 0.87500
Epoch 25/25
2/2 [==============================] - 21s 10s/step - loss:
0.2211 - accuracy: 0.9222 - val_loss: 0.4536 - val_accuracy:
0.8750
```

Validation accuracy is much higher than VGG16 and 19. Let us also look at the training and validation accuracy and the ROC curve:

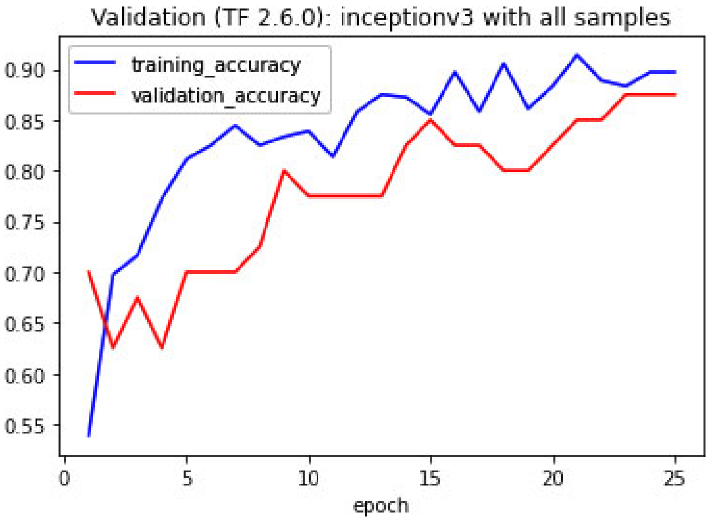


Figure 4.24: *Training and validation accuracy for InceptionV3 based model*

Next, let us look at the ROC curve for this model. It gives an AUC value of about **0.923** as shown in the *figure 4.25*:

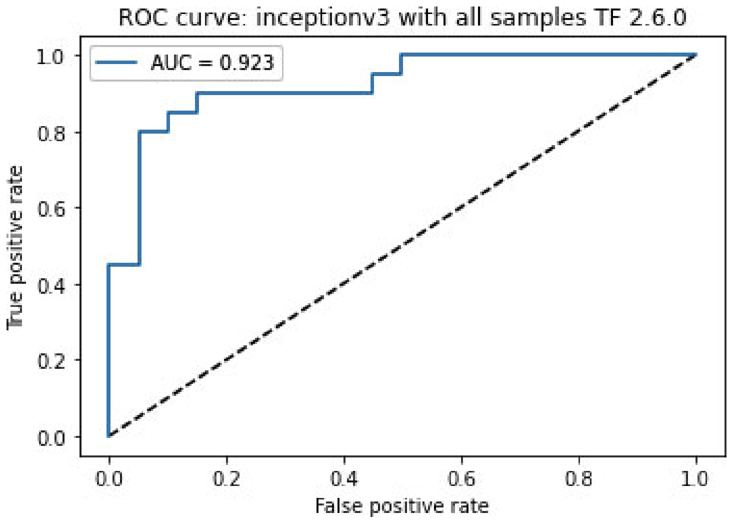


Figure 4.25: *ROC curve for Inception_V3 based model*

With this, we come to the end of the section on advanced image classification techniques. We learnt how to leverage pre-trained models and some of the concepts they apply to achieve those generic models which are or were state of the art.

TensorFlow Hub

Google offers several scripts and tutorials explaining how to directly use its inception networks or how to retrain them for new applications.

The directory dedicated to this architecture in the **`tensorflow/models`** Git repository **https://github.com/tensorflow/hub** is quite well documented.

A pre-trained version of Inception V3 is available on TensorFlow Hub, which gives us the opportunity to introduce this platform. TensorFlow Hub is a repository of pre-trained models.

TensorFlow Hub gives access to pre-trained models so that people do not have to spend time and resources reimplementing and retraining. It combines a website (**https://tfhub.dev**) where developers can search for specific models (depending, for example, on the target recognition task) and a Python package to easily download and start using these models. For example, we can fetch and set up an InceptionV3 network as follows:

```
import tensorflow as tf
import tensorflow as tf
try:
  import tensorflow_hub as hub
```

```
except ModuleNotFoundError:
   print('tensorflow_hub not installed')
url = "https://tfhub.dev/google/tf2-
preview/inception_v3/feature_vector/2"
num_classes = 10
hub_feature_extractor = hub.KerasLayer( # TF-Hub model as Layer
   url, # URL of the TF-Hub model (here, an InceptionV3
   extractor)
   trainable=False, # Flag to set the layers as trainable or not
   input_shape=(299, 299, 3), # Expected input shape (found on
   tfhub. dev)
   output_shape=(2048,), # Output shape (same, found on the
   model's page)
   dtype=tf.float32) # Expected dtype
inception_model = tf.keras.models.Sequential(
   [hub_feature_extractor, tf.keras.layers.Dense(num_classes,
   activation='softmax')],
   name="inception_tf_hub")
inception_model
```

In the preceding example, we create a Keras layer from prebuilt **TF Hub** model and use it in a sequential model.

Next, let us look at a more complete example where we are going to convert a word, a sentence, and a paragraph into a word embedding:

```
# Install the latest Tensorflow version.
!pip3 install --quiet "tensorflow>=2.2"
# Install TF-Hub.
!pip3 install --quiet tensorflow-hub
!pip3 install --quiet seaborn
from absl import logging
import tensorflow as tf
import tensorflow_hub as hub
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import re
import seaborn as sns
```

Load the embeddings file compatible for *TensorFlow 2.x*. An **embedding** is a relatively low-dimensional space into which you can translate high-dimensional vectors:

```
embedding_fn = hub.load("https://tfhub.dev/google/universal-
sentence-encoder/4")
word = "peacock"
sentence = "I am a sentence for which I would like to get its
embedding."
paragraph = (
   "Universal Sentence Encoder embeddings also support short
   paragraphs. "
   "There is no hard limit on how long the paragraph is. Roughly,
   the longer "
   "the more 'diluted' the embedding will be.")
messages = [word, sentence, paragraph]
message_embeddings = embedding_fn(messages)
for i, message_embedding in
enumerate(np.array(message_embeddings).tolist()):
   print("Message: {}".format(messages[i]))
   print("Embedding size: {}".format(len(message_embedding)))
   message_embedding_snippet = ", ".join(
     (str(x) for x in message_embedding[:3]))
   print("Embedding: [{},
   ...]\n".format(message_embedding_snippet))
```

The following output will show the vectors associated with each input:

```
Message: peacock
Embedding size: 512
Embedding: [-0.00719849020242691, -0.004821529611945152,
0.07689614593982697, ...]
Message: I am a sentence for which I would like to get its
embedding.
Embedding size: 512
Embedding: [0.05080859735608101, -0.016524333506822586,
0.01573782041668892, ...]
Message: Universal Sentence Encoder embeddings also support short
paragraphs. There is no hard limit on how long the paragraph is.
Roughly, the longer the more 'diluted' the embedding will be.
Embedding size: 512
```

```
Embedding: [-0.028332691639661 79, -0.0558621920645237,
-0.012941512279212475, ...]
```

You can do more interesting things, like comparing sentence similarity, which we will skip in this example.

Summary

In this chapter, we learnt how to handle machine learning problems using techniques which have been researched like CNN, transfer learning, and so on. We also learnt how to load pre-built models from the TFHub, and use them in sequential models. In transfer learning, we looked at algorithms like VGG16, VGG19, and InceptionV3. You should be able to use TensorFlow to create CNN based models for your own dataset for basic image classification. In the next chapter, we will explore neural network topologies like RNN, LSTM which help in processing sequential data.

Questions

1. What are the advantages of using transfer learning?
2. Why does InceptionV3 have more accuracy network than VGG16?
3. Use TFHub's universal sentence encoder to find similarity between sentences.

References

Very Deep Convolutional Networks for Large-Scale Image Recognition:

https://arxiv.org/abs/1409.1556v4

Code listing

All the examples can be found at the following links:

- **First CNN**:
  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/first_cnn.ipynb
- **CNN basic ops**:

https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/discover_cnns_basic_ops.ipynb

- **TensorFlow 2 ResNet**:

  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/tensorflow2_resnet.ipynb
- **Transfer learning VGG16**:

  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/histopathologic_cancer_detector_vgg16_1000_images.ipynb
- **Transfer learning VGG19**:

  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/histopathologic_cancer_detector_vgg19_1000_images.ipynb
- **Transfer learning inception V3:**

  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/histopathologic_cancer_detector_inceptionv3_1000_images.ipynb
- **TFHub Inception v3**:

  https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/tfhub_inceptionv3.ipynb
- **TF Hub universal sentence encoder**:
  - https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch04/TF_Hub_Universal_Encoder.ipynb
  - https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d

CHAPTER 5

Recurrent Neural Networks

Introduction

RNNs are neural networks suited for sequential data. Sequences of words and time series are some examples of sequence. In these datasets, each time step is related to the previous one. RNNs can also be applied to problems related to computer vision.

We will first start by discussing the basic concepts, and then look at how the weights are learnt.

Structure

In this chapter, we will cover the following topics:

- Basic concepts
- SimpleRNN
    - Building your first SimpleRNN network
    - Weight constraints
    - Text processing with SimpleRNN
- **GRU**
    - Concepts
        - Building your first GRU based model
        - Text processing with GRU
- **LSTM**
    - Concepts
    - Text processing with LSTM
- **Bidirectional LSTM**
    - Concept

- Text processing with bidirectional LSTM

Objectives

In this chapter, the readers will be introduced to the concepts of **Recurrent Neural Network** (**RNN**), **Gated Recurrent Unit** (**GRU**), **Long-Term Short-Term Memory** (**LSTM**), and bi-directional LSTM.

Basic concepts

The **Markov chain model** is a fundamental mathematical construct in statistics and probability, which is used as a building block for modeling sequential patterns via machine learning. We can view our data sequences as *chains*, with each node in the chain dependent in some way on the previous node, so that, *history* is not erased but carried on.

RNN models are also based on this concept of chain structure and vary in how exactly they maintain and update information. Recurrent neural nets apply some form of *loop*. As seen in *figure 5.1*, at some point in time t, the network observes an input x_1 (a word in a sentence) and updates its *state vector* to h_1, from the previous vector h_{1-1}. When we process new input (the next word), it will be done in some manner that is dependent on h_t, and thus, on the history of the sequence (the previous words we've seen affect our understanding of the current word). As seen in *figure 5.1*, this recurrent structure can be viewed as one long unrolled chain, with each node in the chain performing the same kind of processing *step* based on the *message* it obtains from the output of the previous node. This, of course, is very related to the Markov chain models and their **Hidden Markov Model** (**HMM**) extensions, which are not discussed here:

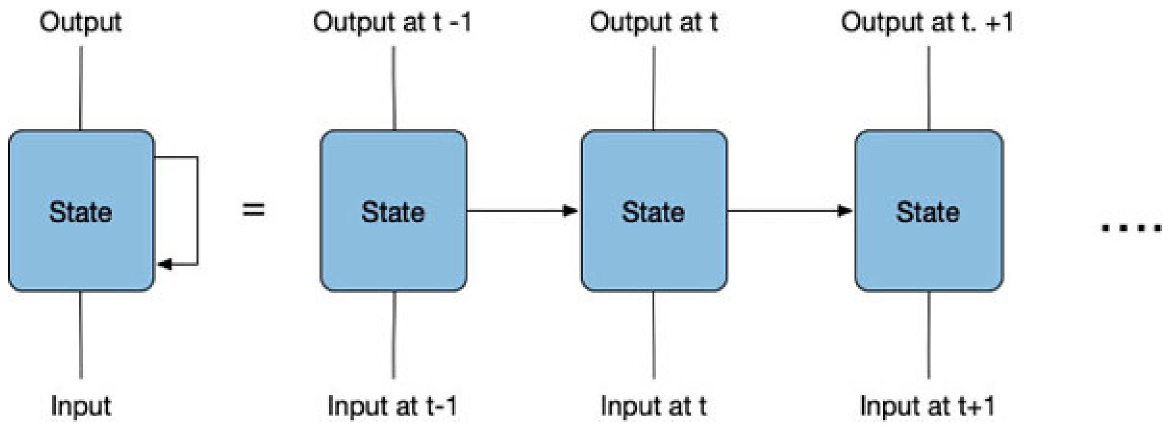


Figure 5.1: *Unfolding of the RNN*

Simple recurrent neural network

It is a type of RNN implemented with the **`classtf.keras.layers.SimpleRNN`** where the output is fed back into the input. It inherits from RNN class: **`tf.keras.layers.RNN`**.

Building your first SimpleRNN network

We will build a simple neural network with a single simple RNN layer with units and input shape as parameters. Units passed to the constructor define the dimensionality of the outer space:

```
import numpy as np
num_samples = 2
timesteps = 3
embedding_dim = 4
units = 2
layer = keras.layers.SimpleRNN(units, input_shape=(None,
embedding_dim))
model = keras.models.Sequential()
model.add(layer)
model.compile('rmsprop', 'mse')
x = np.random.random((num_samples, timesteps, embedding_dim))
y = np.random.random((num_samples, units))
model.train_on_batch(x, y)
```

We created a sequential model with a single RNN layer followed by model compilation and training.

Next, we will look at how to apply constraints to weights and biases; these are useful for overfitting model

Weight constraints

The **`tf.keras.constraints`** module classes allow setting constraints (for example, non-negativity) on model parameters during training. They are per-variable projection functions applied to the target variable after each gradient update when using **`fit()`**. In the following example, we create three different constraints using **`max_norm`**, and apply them as kernel constraints, bias constraints, and recurrent constraints. Detailed explanation for these concepts might be outside the scope of this book:

```
embedding_dim = 4
layer_class = keras.layers.SimpleRNN
k_constraint = keras.constraints.max_norm(0.01)
r_constraint = keras.constraints.max_norm(0.01)
b_constraint = keras.constraints.max_norm(0.01)
layer = layer_class(
    5,
return_sequences=False,
    weights=None,
input_shape=(None, embedding_dim),
kernel_constraint=k_constraint,
recurrent_constraint=r_constraint,
bias_constraint=b_constraint)
layer.build((None, None, embedding_dim))
```

`max_norm` is the implementation used for generating these constraints.

Masking

Masking allows us to handle variable-length inputs in RNNs. Although RNNs can handle variable length inputs, they still need fixed length inputs. Therefore, what we do is create a mask per sample, initialized with **0** with a length equal to the longest sequence in the dataset. Then, we fill the mask with 1s for all the positions where the sample has values. Keras implements a masking class **`keras.layers.Masking`**. The following example shows how masking can be used in a sequential model:

```
layer_class = keras.layers.SimpleRNN
inputs = np.random.random((2, 3, 4))
targets = np.abs(np.random.random((2, 3, 5)))
targets /= targets.sum(axis=-1, keepdims=True)
model = keras.models.Sequential()
model.add(keras.layers.Masking(input_shape=(3, 4)))
model.add(layer_class(units=5, return_sequences=True,
unroll=False))
model.compile(loss='categorical_crossentropy',
optimizer='rmsprop')
model.fit(inputs, targets, epochs=1, batch_size=2, verbose=1)
```

In the later sections, we will look at more real-world applications of masking.

Text processing with SimpleRNN

We are going to use the **Amazon Reviews** dataset for creating a model using RNN to detect sentiment.

This dataset consists of millions of Amazon customer reviews (in the format of input text) and star ratings (as output labels) for learning how to train the model for sentiment analysis.

The idea is to start with a dataset that is more than a toy—real business data on a reasonable scale—but can be trained in minutes on a modest laptop.

We will be using *1000 reviews* to make it an even smaller dataset, so that we can focus on the concepts rather than the accuracy of the model.

Preparing the dataset

Download the dataset from the following link:

https://www.kaggle.com/bittlingmayer/amazonreviews

We have trimmed the dataset to include *1000 reviews*.

Since our code base is in **`Ch05`**, we have defined a few utility methods in **`common_code`**/**`util.py`** which we are planning to use. Follow these steps:

1. Import the appropriate files and define the base variable:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.python.keras import models, layers,
optimizers
import tensorflow
from tensorflow.keras.preprocessing.text import Tokenizer,
text_to_word_sequence
from tensorflow.keras.preprocessing.sequence import
pad_sequences
import bz2
from sklearn.metrics import f1_score, roc_auc_score,
accuracy_score
import re
import os
```

```
print(os.listdir("../data/amazon_reviews"))
base = "../data/amazon_reviews"
```

2. Define a method to get the first *1000 labels,* and text from the file specified. It will return a **`NumPy`** array of labels and text:

```
def get_labels_and_texts(file):
 labels = []
 texts = []
 count = 0
 with open(base + '/train.ft.txt', "r") as a_file:
  for line in a_file:
   if count < 10000:
    #stripped_line = line.strip()
    #x = line.decode("utf-8")
    x = line
labels.append(int(x[9]) - 1)
texts.append(x[10:].strip())
    count = count+1
   else:
     return np.array(labels), texts
```

3. Apply it to **`train.ft.tx`** and **`test.ft.txt`** files:

```
train_labels, train_texts = get_labels_and_texts(base +
'/train.ft.txt')
test_labels, test_texts = get_labels_and_texts(base +
'/test.ft.txt')
```

Let us inspect the **`test_labels`** and **`test_textstrain_labels[0:2]`**. Returns **`array([1, 1])`**and **`train_texts[0:1]`** will return the first value of the text.

Text processing

1. The first thing we need to do to process the text is to write everything in lowercase, and then remove non-word characters. Replace these with spaces, since most of them are going to be punctuation marks. Then, remove any other characters (like letters with accents). It could be better to replace some of these with regular ascii characters, but we are going to ignore that here. If you look at the counts of the different characters, it turns out that there are very few unusual characters in this corpus:

```
from common.util import normalize_texts
```

```
train_texts = normalize_texts(train_texts)
test_texts = normalize_texts(test_texts)
```

Now, we are going to set aside *20%* of the training set for validation:

```
from sklearn.model_selection import train_test_split
train_texts, val_texts, train_labels, val_labels =
train_test_split(
train_texts, train_labels, random_state=57643892,
test_size=0.2)
```

2. Keras provides some tools for converting texts to formats that are useful in deep learning models. We have already done some processing, so now I will just run a Tokenizer using the top **`12000`** words as features:

```
MAX_FEATURES = 12000
tokenizer = Tokenizer(num_words=MAX_FEATURES)
tokenizer.fit_on_texts(train_texts)
train_texts = tokenizer.texts_to_sequences(train_texts)
val_texts = tokenizer.texts_to_sequences(val_texts)
test_texts = tokenizer.texts_to_sequences(test_texts)
```

3. In order to use batches effectively, we will need to take the sequences and turn them into sequences of the same length. We are going to make everything here the length of the longest sentence in the training set. There are also different padding modes that might be useful for different models:

```
MAX_LENGTH = max(len(train_ex) for train_ex in train_texts)
train_texts = pad_sequences(train_texts, maxlen=MAX_LENGTH)
val_texts = pad_sequences(val_texts, maxlen=MAX_LENGTH)
test_texts = pad_sequences(test_texts, maxlen=MAX_LENGTH)
```

4. We will be creating a simple RNN model, which takes in the training data we created earlier and passes it to **`layers.SimpleRNN`**:

```
def build_rnn_model():
 sequences = layers.Input(shape=(MAX_LENGTH,))
 embedded = layers.Embedding(MAX_FEATURES, 64)(sequences)
 x = layers.SimpleRNN(128)(embedded)
 x = layers.Dense(32, activation='relu')(x)
 x = layers.Dense(100, activation='relu')(x)
 predictions = layers.Dense(1, activation='sigmoid')(x)
 model = models.Model(inputs=sequences, outputs=predictions)
model.compile(
   optimizer='rmsprop',
  loss='binary_crossentropy',
  metrics=['binary_accuracy']
 )
 return model
rnn_model = build_rnn_model()
```

5. We will now train the model on the training text and labels and validate on validation text and labels:

```
history = rnn_model.fit(
train_texts,
train_labels,
batch_size=128,
 epochs=10,
validation_data=(val_texts, val_labels), )
```

6. We are able to achieve a validation accuracy of `0.81`, which is quite decent for a simple RNN based model:

```
Epoch 10/10
8000/8000 [==============================] - 19s 2ms/sample
- loss: 0.1437 - binary_accuracy: 0.9489 - val_loss: 0.6114
- val_binary_accuracy: 0.8100
```

7. Let us plot the accuracy and loss:

```
from common.plot_util import eval_metric
eval_metric(rnn_model, history, 'loss', 10)
```

   Validation loss goes up again after a few epochs giving an indication that the model is overfitted, as shown in *figure 5.2*:

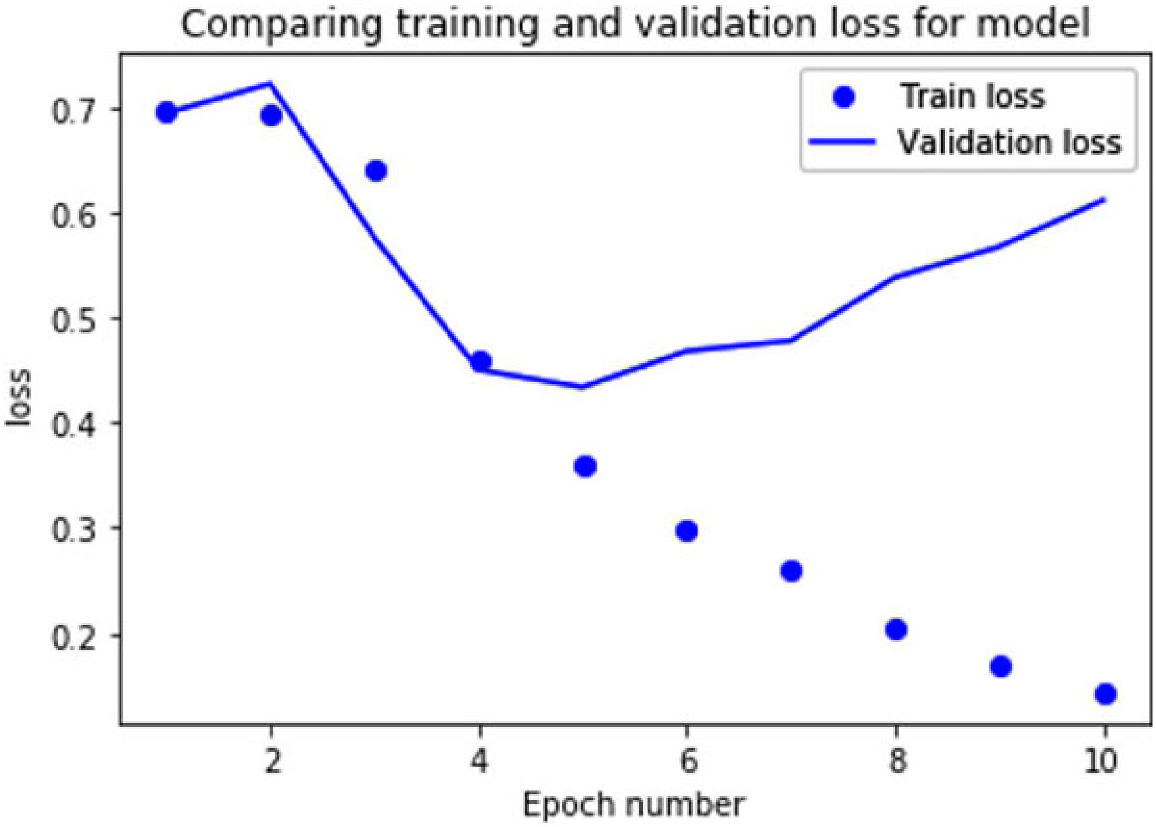


Figure 5.2: *Training and validation loss for simple RNN based model to predict sentiment of Amazon reviews*

8. Let us plot the accuracy for training and validation data. Validation accuracy tapers off at `0.8` after *epoch 3,* as illustrated in *figure 5.3*:

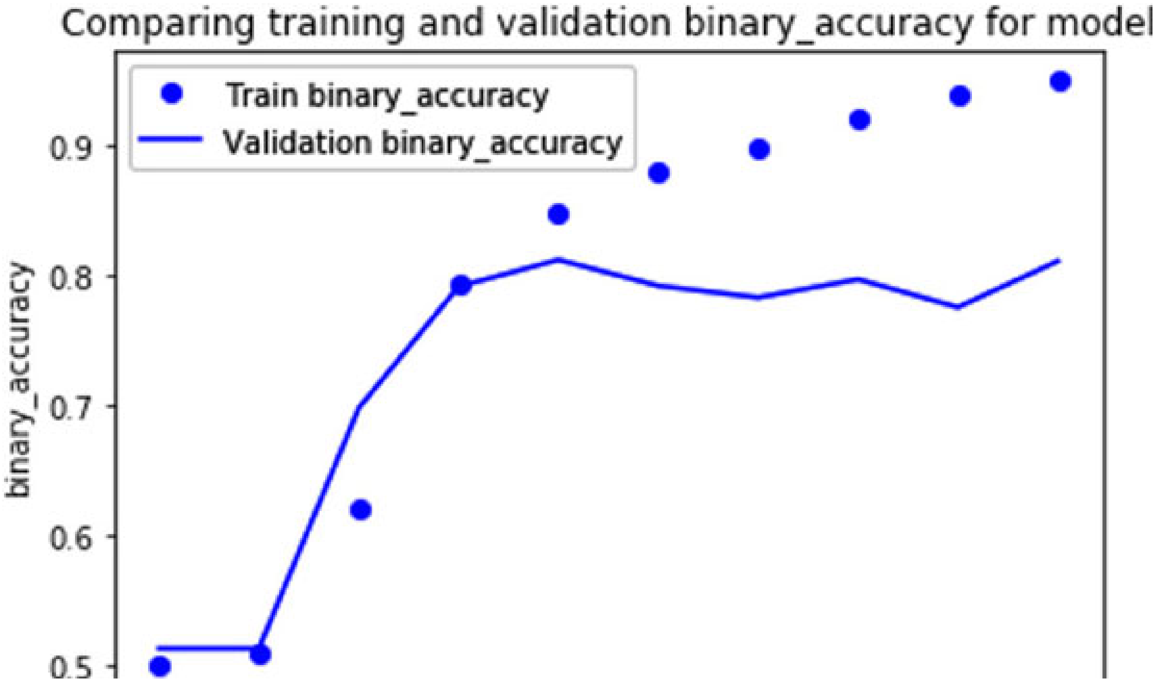


Figure 5.3: *Training and validation accuracies for simple RNN based model to predict sentiment of Amazon reviews*

We learnt how to implement a basic RNN model in TensorFlow. Next, we will look at a more sophisticated model. Notice how the validation loss actually increases, which is a clear sign of overfitting. We will look at the next section and how this can be handled.

Gated recurrent unit

Gated Recurrent Unit (**GRU**), introduced by *Cho, et al.* in *2014,* solves the vanishing gradient problem found in a standard recurrent neural network. GRU is a variation on the LSTM because both are designed similarly.

GRUs are an improved version of standard recurrent neural network.

To solve the vanishing gradient problem of a standard RNN, GRU uses the so-called **update gate** and **reset gate**. These are the two vectors that decide what information should be passed to the output. The special thing about them is that they can be trained to save information from long ago. *Figure 5.4* illustrates the input and outputs of GRU cells in a sequence:

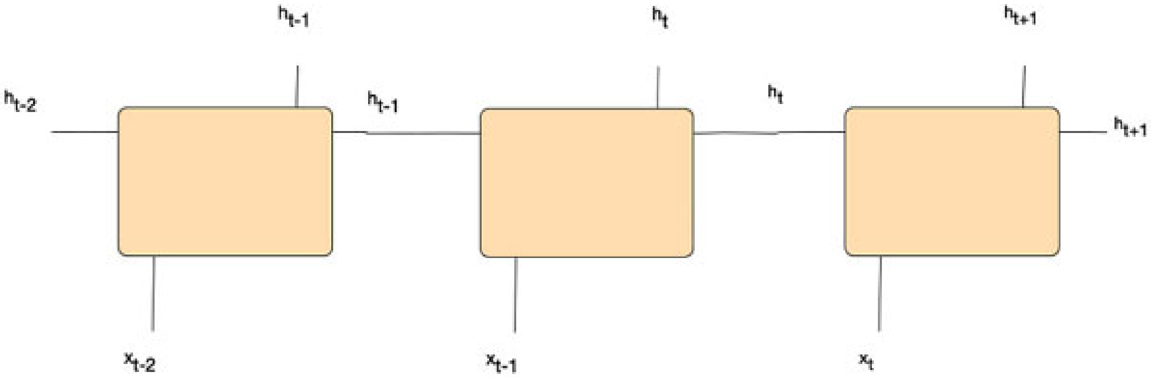


Figure 5.4: *GRU based cells with inputs and outputs in a sequence*

To explain the mathematics behind it, we will examine a single unit from the following recurrent neural network, shown in *figure 5.5*:

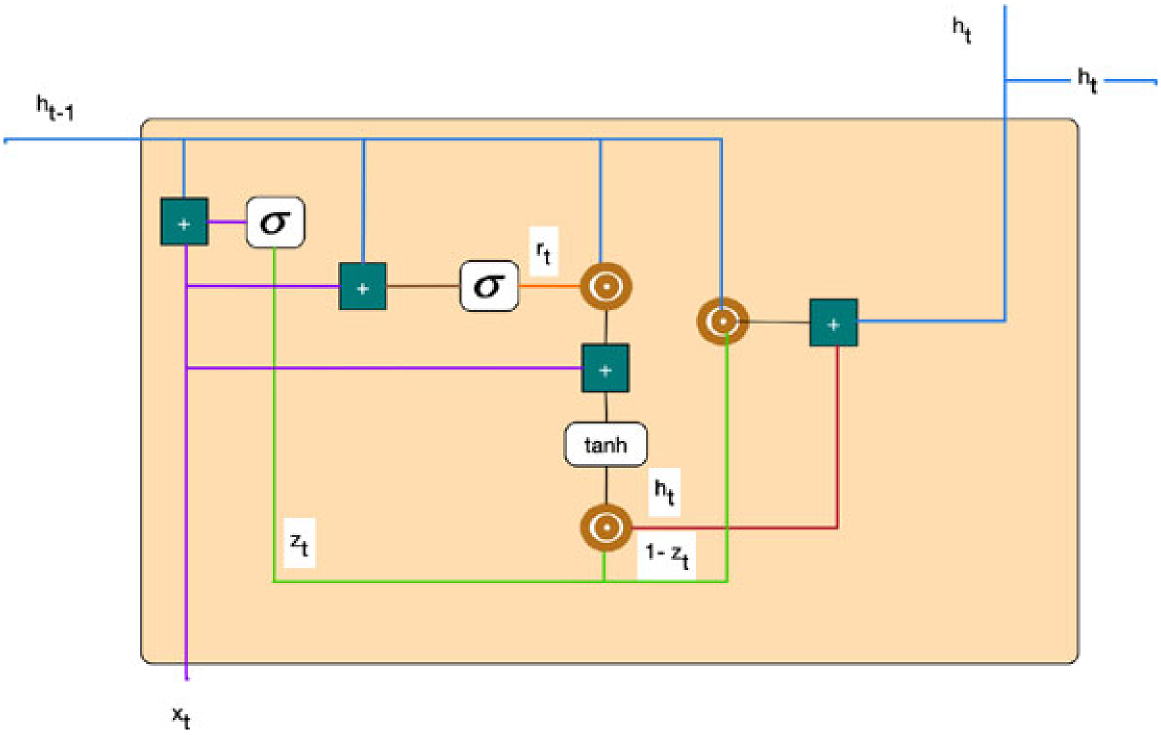


Figure 5.5: *GRU cell*

Let us first introduce the notations as shown in *figure 5.6*:

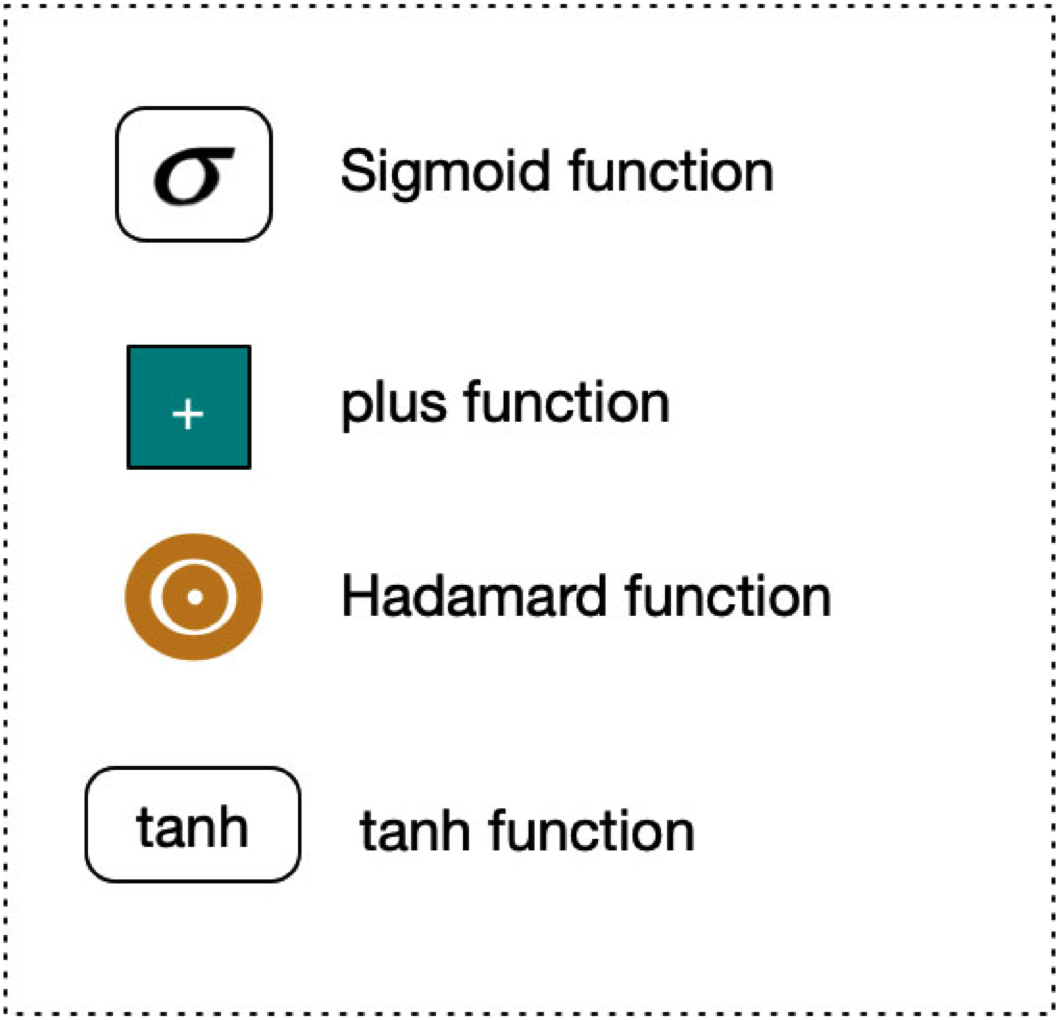


Figure 5.6: *GRU based cells notation*

Update gate

We will start by explaining the update. We will first calculate the update gate *zt* for time step *t* using the following formula:

$$Z_t = S(W^{(z)} x_t + U^{(z)} h_{t-1})$$

Equation explaining the output of update gate

When x_t is plugged into the network unit, it is multiplied by its own weight $W^{(z)}$. The same happens for h_{t-1}, which holds the information about the previous *t-1* units. It is multiplied by its own weight $U^{(z)}$. Both the results are added together, and a sigmoid activation function is applied to obtain values between **0** and **1**. Following the preceding schema, we have the given illustration:

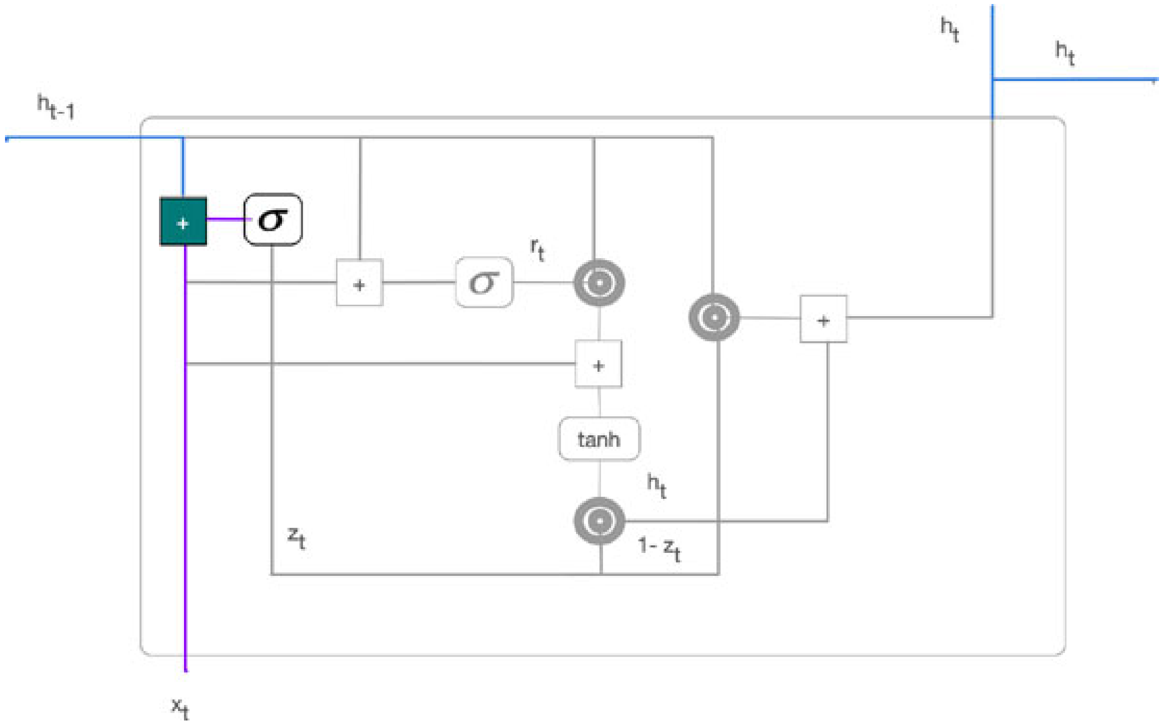


Figure 5.7: *GRU: elements of update gate*

Reset gate

Primarily, this gate is used by the model to decide how much of the past information to forget. To calculate it, we use the following formula:

$$r_t = S(W^{(r)}\, x_t + U^{(r)}\, h_{t-1})$$

This formula is the same as the one for the update gate. The difference comes in the weights and the gate's usage. The following schema shows where the reset gate is:

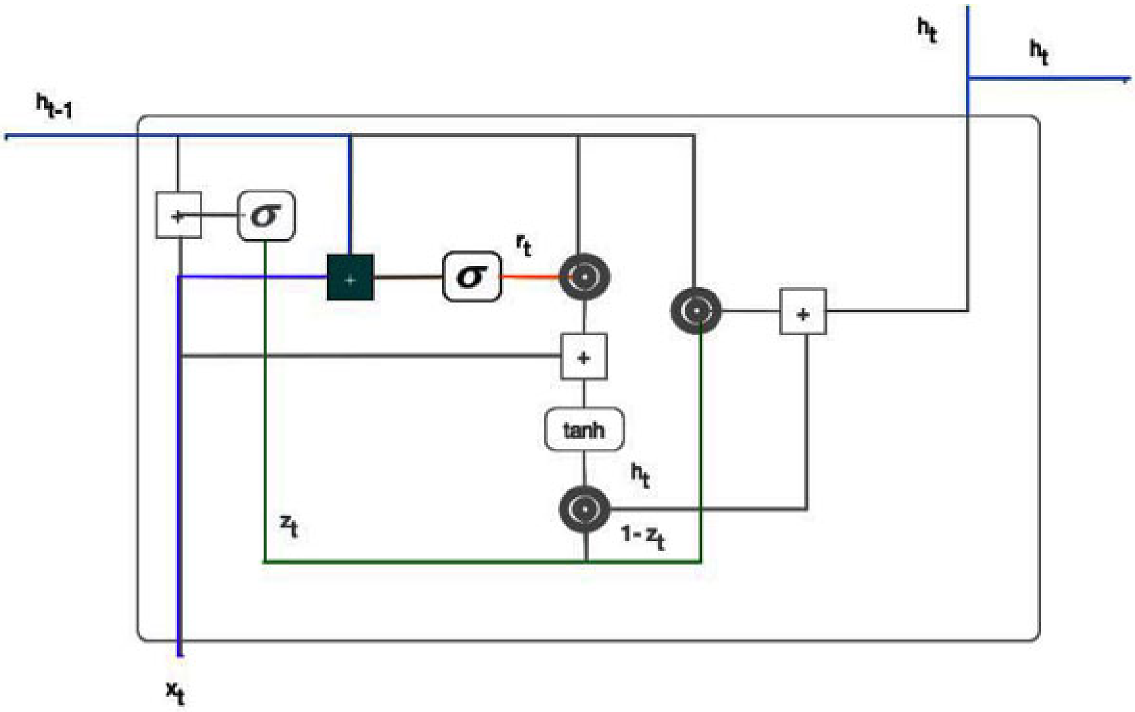


Figure 5.8: *GRU: elements of reset gate*

Current memory content

Next, we will explore how the gates affect the final output. We will start with exploring how the reset gate is used. A new memory content is introduced, which will use the reset gate to store the relevant information from the past. It is calculated as follows:

1. Multiply the input x_t with a weight W and h_{t-1} with a weight U.
2. Calculate the **Hadamard (elementwise)** product between the reset gate r_t and Uh_{t-1}. This will determine the data to be removed from the previous time steps. Say, we want to do a sentiment analysis on a review and the negative sentiment is only the last few words of the review. To determine the overall level of the sentiment from the book, we will only need the last part of the review. As the neural network goes to the end of the text, it will learn to assign r_t vector, close to *0*, ignoring the past and focusing only on the last sentences.
3. Next, we will sum up the results of *Step 1* and *Step 2*.
4. In the end, apply the nonlinear activation function `tanh`.

Figure 5.9 illustrates various elements on the memory unit in GRU:

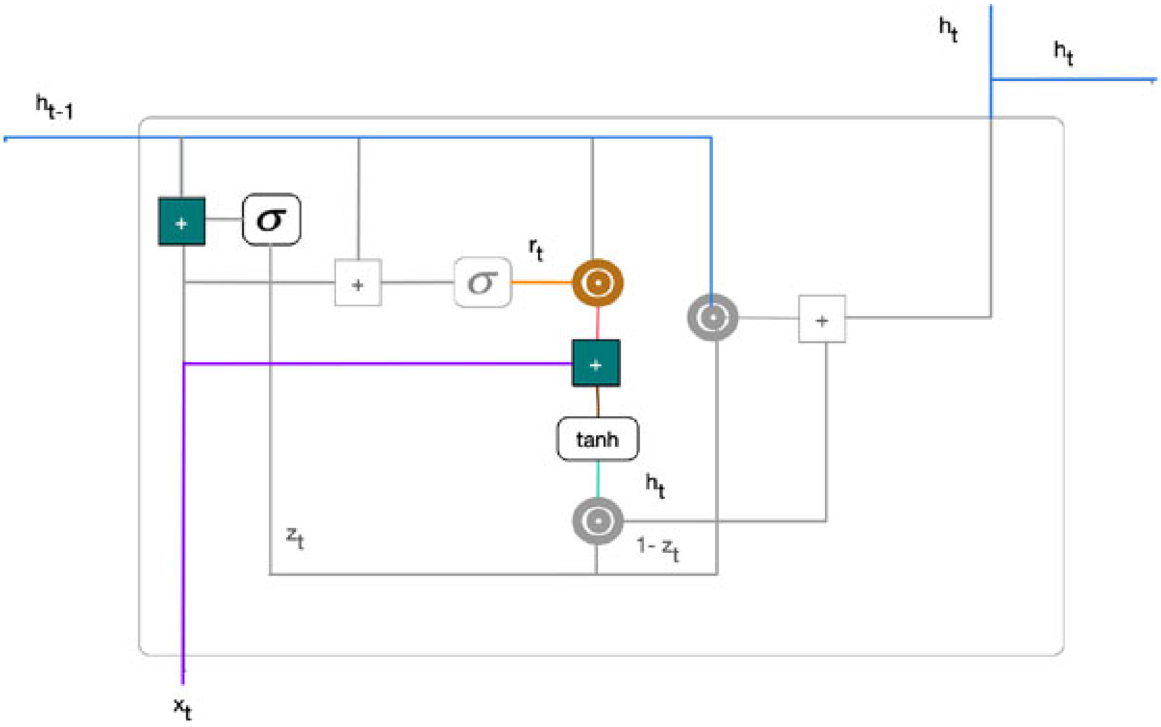


Figure 5.9: *Elements of memory unit*

We looked at the various elements of a GRU network. Next, let us look at the implementation in TensorFlow 2.7.

GRU implementation in TensorFlow 2.7

In *TensorFlow 2.7*, GRU is implemented under the API **`tf.layers.keras.GRU`**. It is a subclass of RNN and an earlier version of GRU. *Figure 5.10* illustrates the GRU implementation in TensorFlow:

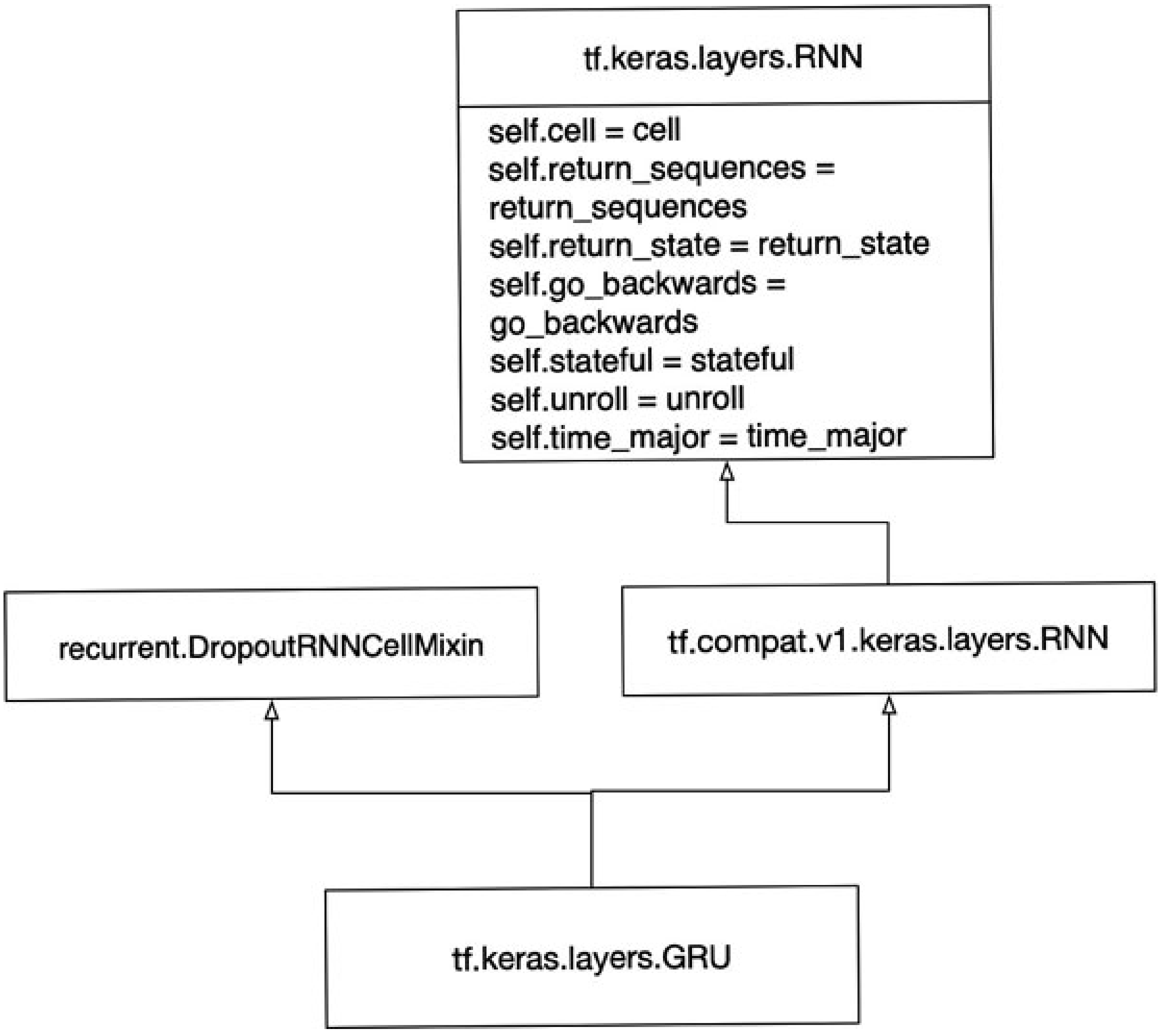


Figure 5.10: *GRU implementation in TensorFlow*

In the next section, we will build a GRU-based model for test data using TensorFlow APIs.

Building your first GRU-based model

We will be building the first GRU-based model and comparing the accuracy with the simple RNN:

1. Make the relevant import and define constants:

```
from tensorflow.python.keras import testing_utils
import warnings
warnings.filterwarnings('ignore')
from tensorflow.python import keras
import numpy as np
```

```
from tensorflow.python.keras.utils import np_utils
from tensorflow.python.keras.layers import recurrent_v2 as
rnn
from tensorflow.python.framework import dtypes
```

2. Notice the **`rnn_state_size`**:

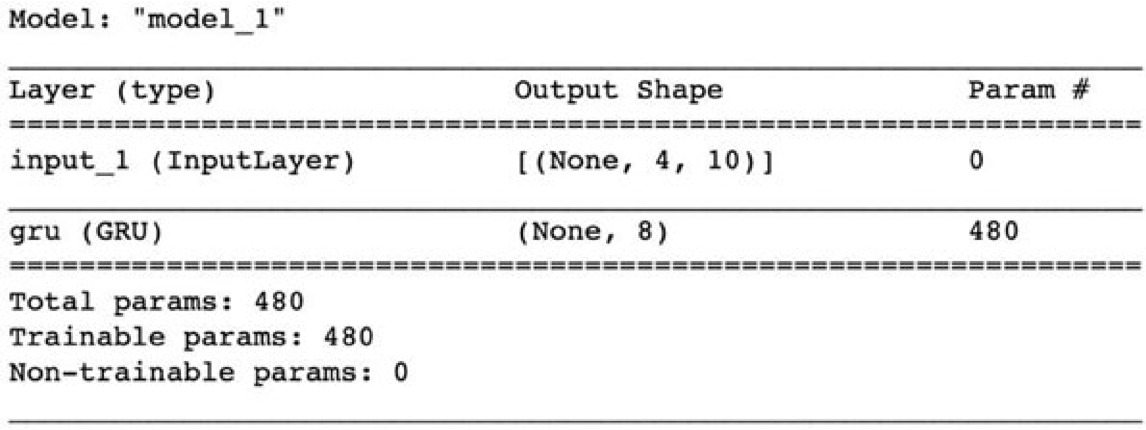
```
batch = 100
timestep = 4
input_shape = 10
output_shape = 8
rnn_state_size = 8
epoch = 10
```

3. Create input and output layers with GRU with appropriate **`state_size`**:

```
(x_train, y_train), _ = testing_utils.get_test_data(
train_samples=batch,
test_samples=0,
input_shape=(timestep, input_shape),
num_classes=output_shape)
y_train = np_utils.to_categorical(y_train, output_shape)
layer = rnn.GRU(rnn_state_size)
inputs = keras.layers.Input(
  shape=[timestep, input_shape], dtype=dtypes.float32)
outputs = layer(inputs)
```

4. Let us create a model with a single GRU layer as the output layer:

```
model = keras.models.Model(inputs, outputs)
model.summary()
```



```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 4, 10)]           0
_________________________________________________________________
gru (GRU)                    (None, 8)                 480
=================================================================
Total params: 480
Trainable params: 480
Non-trainable params: 0
_________________________________________________________________
```

5. Compile and train the model:

```
model.compile('rmsprop', loss='mse')
```

```
    model.fit(x_train, y_train, epochs=epoch)
    model.evaluate(x_train, y_train)
    predictions = model.predict(x_train)
```

We got a basic idea of the APIs exposed for GRU implementation. Let us look at a more real-life example in the next section.

Text processing with GRU

We will be using the same dataset we created earlier, and use GRU instead of **SimpleRNN**.

The dataset is the Amazon reviews' dataset found at the following link:

https://www.kaggle.com/muonneutrino/sentiment-analysis-with-amazon-reviews

Please refer to the previous sections on how to load the dataset. We will create a model that uses two layers of **`tf.keras.layers.GRU`** after the input layers and the embedded layer:

```
def build_rnn_model():
   sequences = layers.Input(shape=(MAX_LENGTH,))
   embedded = layers.Embedding(MAX_FEATURES, 64)(sequences)
   x = layers.GRU(128, return_sequences=True)(embedded)
   x = layers.GRU(128)(x)
   x = layers.Dense(32, activation='relu')(x)
   x = layers.Dense(100, activation='relu')(x)
   predictions = layers.Dense(1, activation='sigmoid')(x)
   model = models.Model(inputs=sequences, outputs=predictions)
model.compile(
    optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['binary_accuracy']
   )
   return model
rnn_model = build_rnn_model()
Next let us train this model:
history = rnn_model.fit(
train_texts,
train_labels,
batch_size=128,
   epochs=10,
validation_data=(val_texts, val_labels), )
```

It will give a validation accuracy of around `0.834`, which is better than what we achieved with simple RNN (`0.81`):

```
Epoch 10/10
8000/8000 [==============================] - 125s 16ms/sample -
loss: 0.0748 - binary_accuracy: 0.9786 - val_loss: 0.7409 -
val_binary_accuracy: 0.8430
```

Let us look at the plots as a function of epochs (refer to *figure 5.11*):

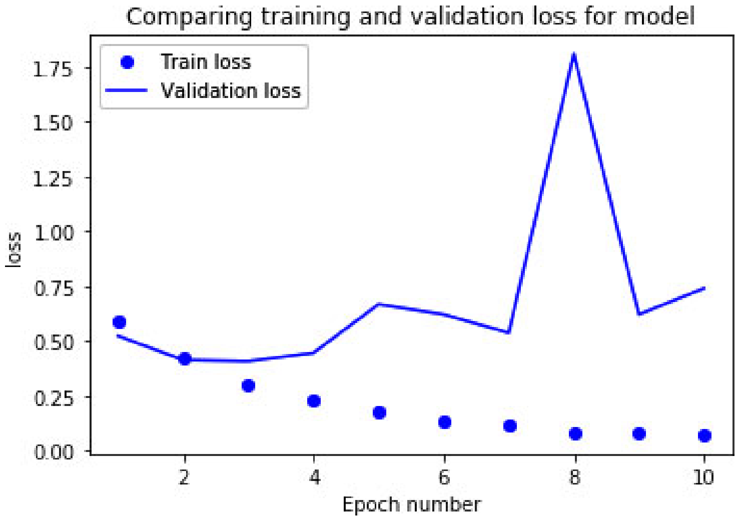


Figure 5.11: *Training and validation loss for GRU based model to predict sentiment of the Amazon reviews*

Training and validation accuracy for text processing with the GRU model is shown in *figure 5.12*:

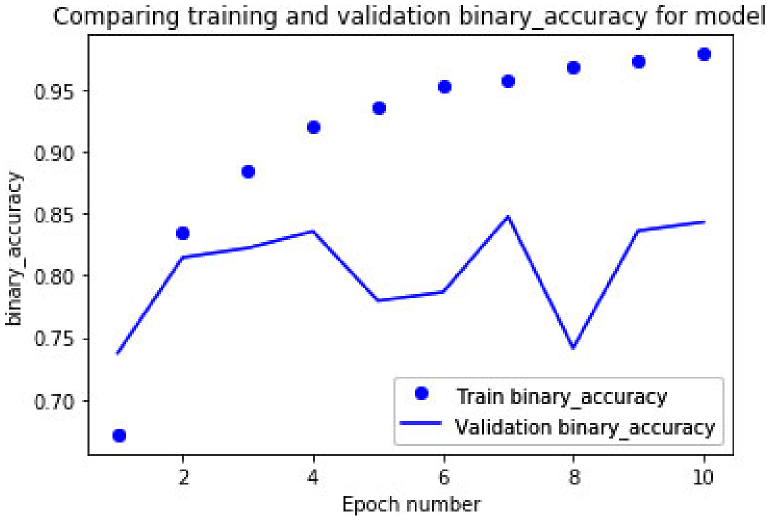


Figure 5.12: *Training and validation accuracy for GRU based model to predict sentiment of Amazon reviews*

GRU gives a binary accuracy of `0.843,` as compared to `0.81` by simple RNN. In the next section, we will look at another technique to capture the patterns in memory as data moves through the RNN network.

Long-Term Short-Term Memory (LSTM)

A **Long-Term Short-Term Memory** (**LSTM**) has a controlled flow like a recurrent neural network. It processes data by passing on the information as the data propagates forward. The differences are in the operations within the LSTM's cells.

LSTM cell operations

These operations are used to allow the LSTM to keep or forget information. We will go over this step-by-step.

Let us first look at a LSTM cell and its components, refer to *figure 5.13*:

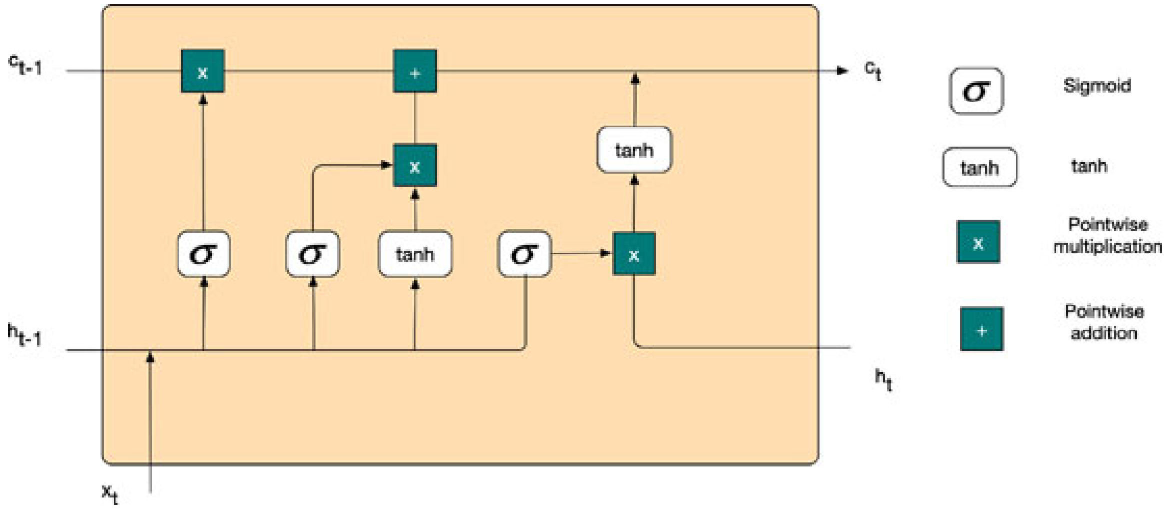


Figure 5.13: *LSTM cell*

LSTM is made up of three gates–**forget gate**, **input gate**, and **output gate**.

Figure 5.14 illustrates the elements of a *forget gate:*

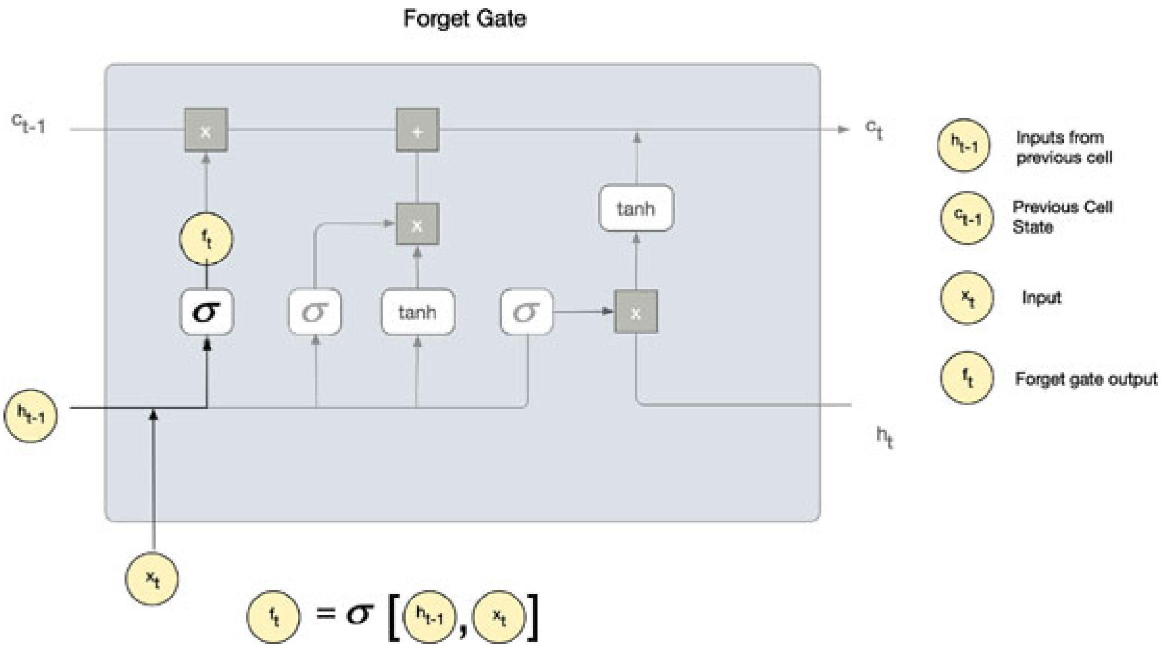


***Figure 5.14:** LSTM cell forget gate*

- First, sigmoid function in a LSTM cell has a **forget gate**. This gate is used to decide what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between **0** and **1**. The closer to **0** means to forget, and the closer to **1** means to keep.
- Next, the sigmoid and tanh functions are referred to as the **input gate,** as shown in *figure 5.15*:

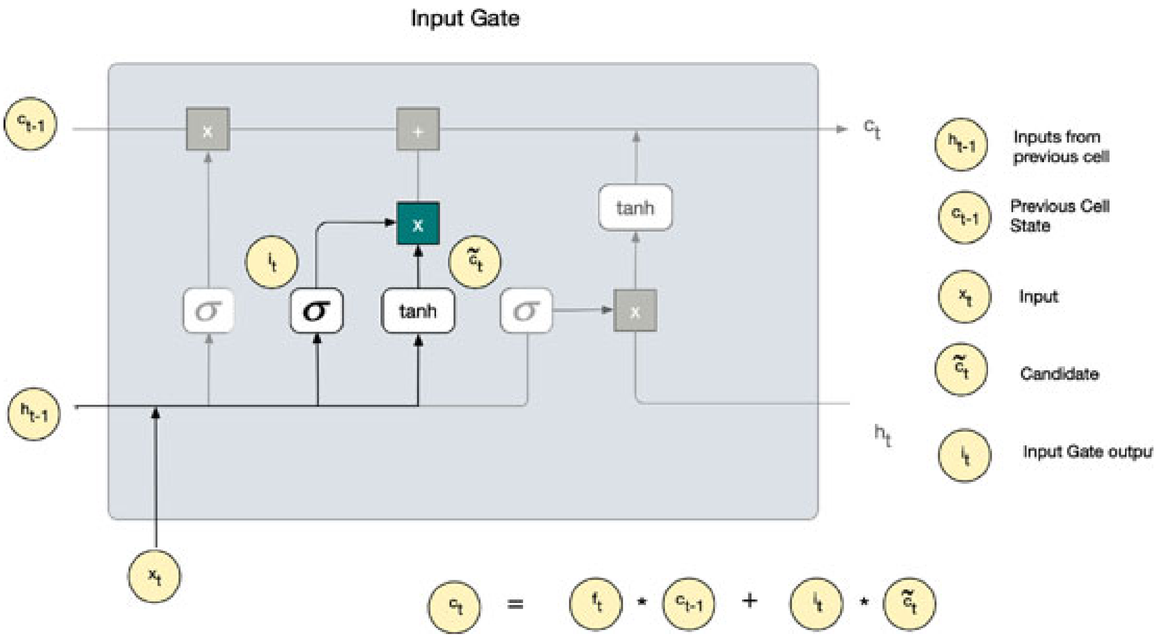


Figure 5.15: *LSTM cell input gate*

- The last sigmoid function is the **output gate,** and it highlights which information should be going to the next **hidden state**, refer to *figure 5.16* for illustration:

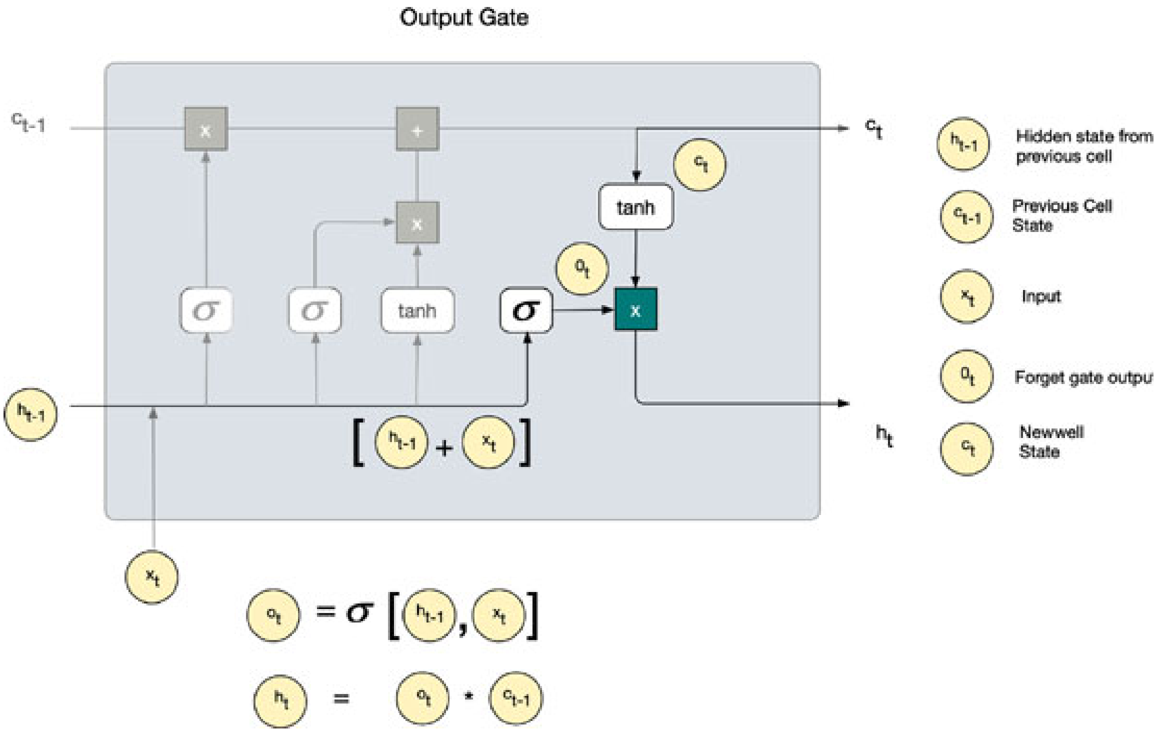


Figure 5.16: *LSTM cell output gate*

The hidden state has information on previous inputs. The hidden state is also used to make predictions. First, we pass the previous hidden state and h_{t-1} the current input into a sigmoid function. Next, we pass the newly modified cell state c_t to the **`tanh`** function. The **`tanh`** output is multiplied with the sigmoid output to decide what information the hidden state h_t should carry. The output is the hidden state h_t. The new cell state c_t and the new hidden h_t is then carried over to the next time step.

Text processing with LSTM

TensorFlow provides class layers. LSTM can be used to create an LSTM layer as part of the sequence model or function model.

Let us look at the same Amazon reviews' dataset, and process it using the LSTM based model:

1. First, we will build the LSTM model with two layers:

   ```
   def build_rnn_model():
    sequences = layers.Input(shape=(MAX_LENGTH,))
    embedded = layers.Embedding(MAX_FEATURES, 64)(sequences)
    #x = layers.CuDNNGRU(128, return_sequences=True)(embedded)
    #x = layers.CuDNNGRU(128)(x)
    x = layers.LSTM(128, return_sequences=True)(embedded)
    x = layers.LSTM(128)(x)
    x = layers.Dense(32, activation='relu')(x)
    x = layers.Dense(100, activation='relu')(x)
    predictions = layers.Dense(1, activation='sigmoid')(x)
    model = models.Model(inputs=sequences, outputs=predictions)
    model.compile(
     optimizer='rmsprop',
     loss='binary_crossentropy',
     metrics=['binary_accuracy']
    )
    return model
   rnn_model = build_rnn_model()
   ```

2. Train the model and observe the accuracy and loss values:

   ```
   history = rnn_model.fit(
   train_texts,
   train_labels,
   ```

```
batch_size=128,
 epochs=10,
validation_data=(val_texts, val_labels), )
```

3. Training output will give the final accuracies and losses:

```
Epoch 10/10
8000/8000 [==============================] - 126s
16ms/sample - loss: 0.0382 - binary_accuracy: 0.9884 -
val_loss: 0.7318 - val_binary_accuracy: 0.8430
```

Let us plot the losses and accuracies. Notice how the validation loss still keeps going up in the *figure 5.17*:

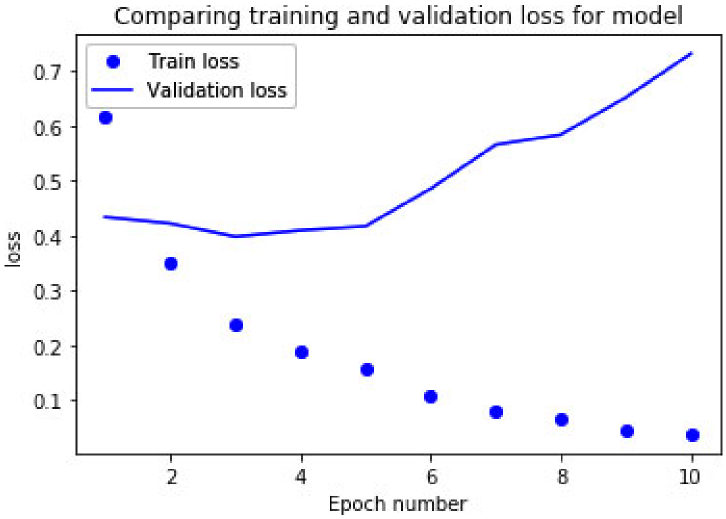


Figure 5.17: *Training and validation accuracy for Amazon review sentiment prediction with LSTM-based network*

On the other hand, validation accuracy tapers off after three iterations as shown in *figure 5.18*:

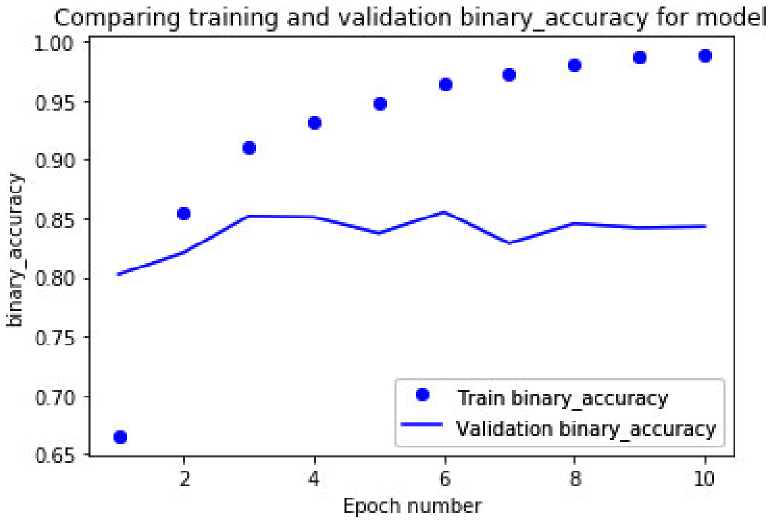


Figure 5.18: *Training and validation loss for Amazon review sentiment prediction with LSTM based network*

The model is clearly overfitting, which can be handled by various techniques. While the accuracy is higher than **SimpleRNN** or **GRU** based network, there is still scope for improving this further.

Bidirectional LSTM

Bidirectional Recurrent Neural Networks (**BRNN**) are a type of network which connect two hidden layers of opposite directions to the same output. The output layer can get information from past (backwards) and future (forward) states, simultaneously. Invented in *1997* by *Schuster* and *Paliwal* (**https://en.wikipedia.org/wiki/Bidirectional_recurrent_neural_networks#cite_note-Schuster-1**) [1], BRNNs were introduced to increase the input data available to the network.

Text processing with bidirectional LSTM

Next, we will use a variation of LSTM called **bidirectional LSTM**:

```
def build_rnn_model():
   sequences = layers.Input(shape=(MAX_LENGTH,))
   embedded = layers.Embedding(MAX_FEATURES, 64)(sequences)
   #x = layers.CuDNNGRU(128, return_sequences=True)(embedded)
   #x = layers.CuDNNGRU(128)(x)
   x = layers.Bidirectional(layers.LSTM(128))(embedded)
   #x = layers.LSTM(128, return_sequences=True)(embedded)
   #x = layers.LSTM(128)(x)
   x = layers.Dense(32, activation='relu')(x)
   x = layers.Dense(100, activation='relu')(x)
   predictions = layers.Dense(1, activation='sigmoid')(x)
   model = models.Model(inputs=sequences, outputs=predictions)
model.compile(
    optimizer='rmsprop',
    loss='binary_crossentropy',
    metrics=['binary_accuracy']
   )
   return model
rnn_model = build_rnn_model()
```

Output from the model training:

```
Epoch 10/10
8000/8000 [==============================] - 108s 14ms/sample -
loss: 0.0426 - binary_accuracy: 0.9864 - val_loss: 1.2382 -
val_binary_accuracy: 0.8040
```

Let us look at the plots for accuracy, and loss for training and validation set as illustrated in *figure 5.19*:

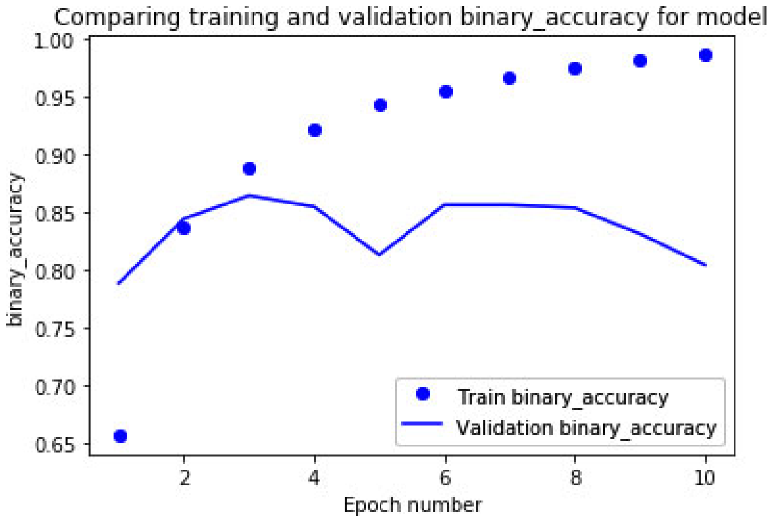


Figure 5.19: *Training and validation loss for Amazon review sentiment prediction with bidirectional LSTM based network*

Final validation accuracy of `0.8040` is actually lower than plain LSTM-based model.

Next, we will look at the loss for the model, refer to *figure 5.20*, where training and validation loss for the LSTM model is plotted:

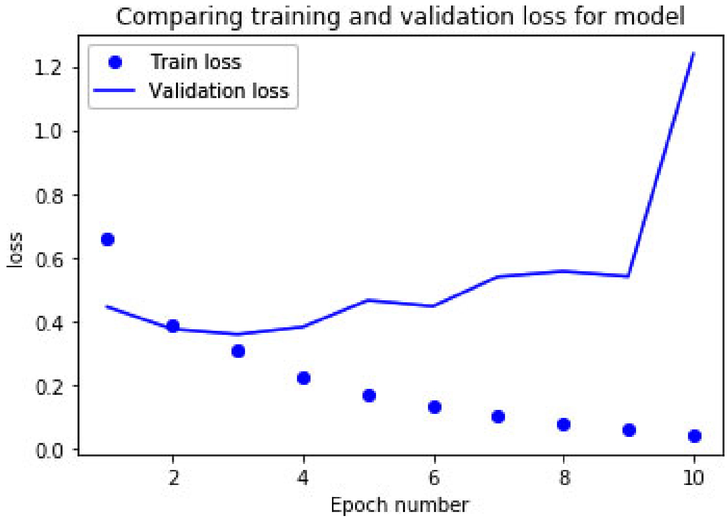


Figure 5.20: *Training and validation loss for Amazon review sentiment prediction with bidirectional LSTM-based network*

As can be seen with **128 units**, the network is clearly overfitting for bidirectional LSTM as well.

Conclusion

In this chapter, we learnt the concept of recurrent neural network and how it is used to process sequence-based datasets. We started with the basic concept of chain structure behind RNN. We looked at how TensorFlow exposes various flavors of RNN– **SimpleRNN**, **GRU**, and **LSTM**. We applied these specific flavors to the Amazon reviews' dataset, and compared the results.

This forms the basis from which the reader can expand and apply RNN to more varied problems in this domain.

Points to remember

- RNN is used for processing input which has a sequence of steps.
- SimpleRNN is the simplest form of RNN supported by TensorFlow.
- GRU has reset gate, update gate, and memory cell.
- LSTM has three gates–forget, input, and output gate.
- LSTM uses **Sigmoid** function for the forget gate.

Multiple choice questions

1. **SimpleRNN has the following gate/s:**
    a. Tanh function
    b. Sigmoid function
    c. none of these
    d. a and b
2. **GRU has a forget gate:**
    a. true
    b. false
3. **LSTM is considered a better performant gate because:**
    a. it can perform unbounded operations.
    b. It doesn't have a forget gate.
    c. none of the above

Answers

1. **d**
2. **b**
3. **c**

Questions

1. How is GRU different from SimpleRNN?
2. What are the pre-processing steps for a text sequence to be processed by RNN?

Key terms

- **RNN**: Recurrent Neural Networks
- **LSTM**: Long-Term Short-Term Memory
- **GRU**: Gated Recurrent Unit
- Sigmoid function

CHAPTER 6

Time Series Forecasting with TensorFlow

Introduction

We have learnt in the preceding chapters how TensorFlow can be used to process various kinds of datasets. Topology of the networks varies based on the type of dataset and the use case. We have not covered the time series data, and how to handle. This chapter focusses on this special kind of dataset.

Structure

In this chapter, we will discuss the following topics:

- Machine learning and time series
- Types of time series
- Generating synthetic time series dataset
- Using basic deep neural network to predict a time series dataset
- RNN and how it can be used to predict a time series dataset
- LSTM and time series

Objectives

After studying this chapter, you will be able to generate a time series dataset. You will also be able to create various types of models to be able to predict next value in the series based on the generated synthetic dataset. We will start with the different types of time series data. This will be followed by sections explaining the generation of time series data using TensorFlow utilities. Then, we will look at using single and multi-layer feedforward network on the generated dataset. In the end, we will look at how network types like RNN and LSTM help improve accuracy of predictions.

Background

Time series datasets are everywhere. They are part of our everyday life. Let us take an example of daily sugar levels of diabetes patients or weekly readings of blood pressure. *Figure 6.1* plots **Date versus Pre-Breakfast Sugar Levels,** which is a perfect time series data example:

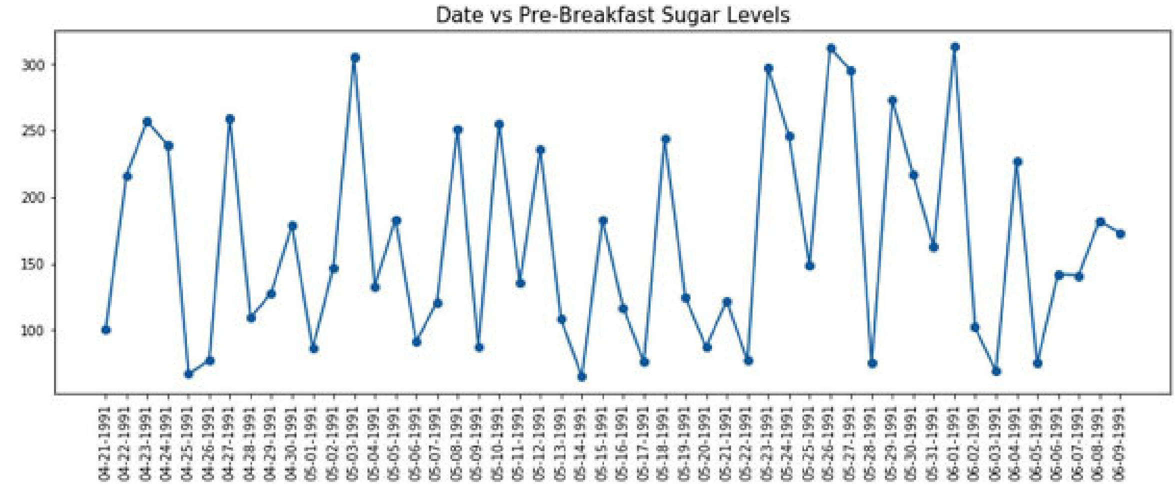


Figure 6.1: *Example of a time series data – sugar level of a diabetes patient as a function of time*

Reference link is as follows:

https://www.joeniekrofoundation.com/stroke-2/3685/attachment/diabetes-blood-sugar-chart/

Time series could also be multi-variate, where the target variable is more than one. This can help understand how two dependent variables change as a function of time. Let us plot a multi-variate analysis of per capita income of various countries. *Figure 6.2* provides an example of multi-variate time series. In this case, per capita income of various countries has been plotted as a function of year:

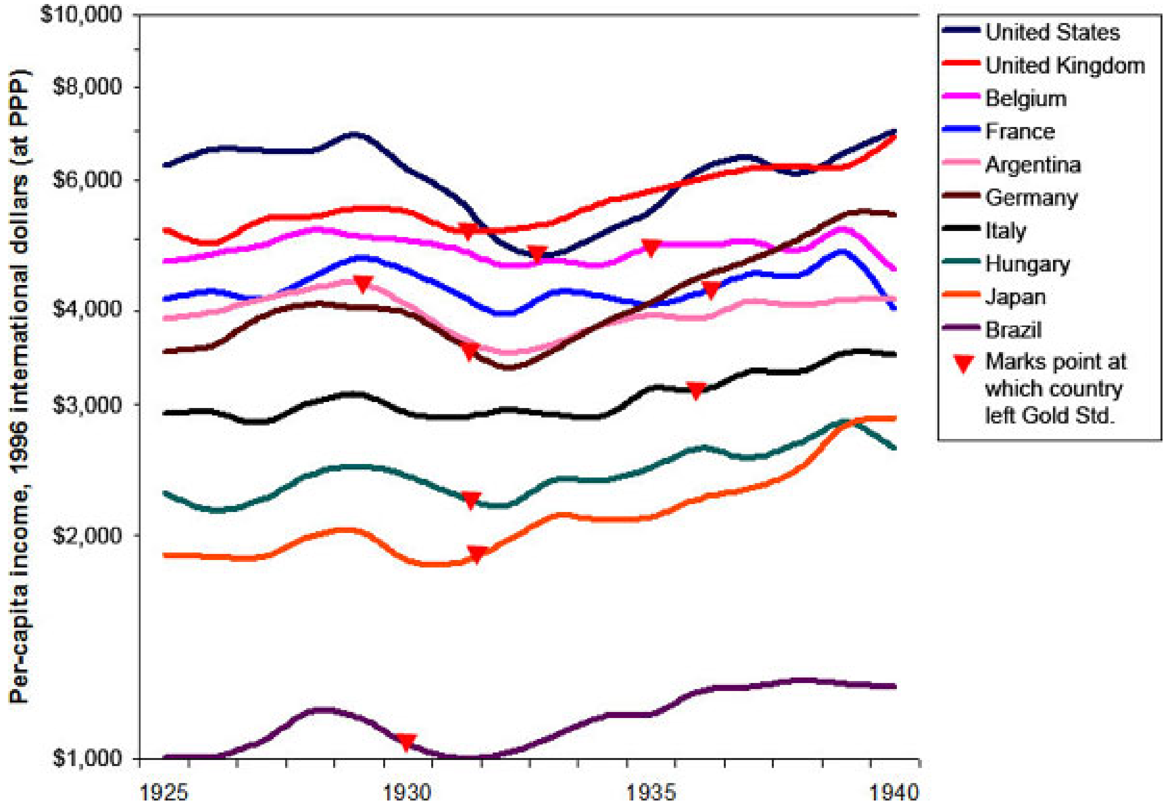


Figure 6.2: *Example of a time series data-per capita income for various countries over a period of 15 years*

Image (**https://en.wikipedia.org/wiki/File:International_depression.png**) available under **https://creativecommons.org/licenses/by-sa/3.0/** on *Wikipedia*.

Machine learning and time series

The most obvious application of machine learning for the time series dataset is forecasting a value in the future. It can be used to predict value in the past, or *impute* a missing value between two data points in an existing dataset. It can also be used to detect anomalies in a dataset. It can be used to analyze patterns in speech and detect words from that.

Common patterns in time series

Some common patterns found in time series data are:

- **Trend**: have a specific direction, it moves in for example *Moore's law*.

- **Seasonality**: recurring patterns in predictable interval, for example, shopping sites sales peaking on weekends.
- **White noise**: totally random instance in an otherwise predictable time series dataset.
- **Non-stationary time series**: time series that changes its direction because of a single event.

First time series notebook with synthetic data

Let us look at the features of **`tf.dataset`**, which we are going to use to create a synthetic dataset:

1. We will start by making the imports of **`tensorflow`** and **`numpy`** libraries:

   ```
   import tensorflow as tf
   import numpy as np
   ```

2. We will the create a dataset with range **`10`**:

   ```
   dataset = tf.data.Dataset.range(10)
   for val in dataset:
    print(val.numpy())
   ```

   Output will produce a range:

   ```
   0
   1
   2
   3
   4
   5
   6
   7
   8
   9
   ```

3. Next, we will create a dataset with **`window`** of **`5`** and shifted by **`1`**:

   ```
   dataset = tf.data.Dataset.range(10)
   dataset = dataset.window(5, shift=1)
   for window_dataset in dataset:
    for val in window_dataset:
    print(val.numpy(), end=" ")
    print()
   ```

   With the following output, you can see that the outputs are not complete for iterations:

   ```
   0 1 2 3 4
   1 2 3 4 5
   2 3 4 5 6
   3 4 5 6 7
   4 5 6 7 8
   ```

```
5 6 7 8 9
6 7 8 9
7 8 9
8 9
9
```

Drop outputs which are not complete:

```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
for window_dataset in dataset:
 for val in window_dataset:
 print(val.numpy(), end=" ")
 print()
```

This gives the truncated outputs as follows:

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
```

4. Next, we will create dependent and independent variables:

```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
dataset = dataset.map(lambda window: (window[:-1],
window[-1:]))
for x,y in dataset:
 print(x.numpy(), y.numpy())
[0 1 2 3] [4]
[1 2 3 4] [5]
[2 3 4 5] [6]
[3 4 5 6] [7]
[4 5 6 7] [8]
[5 6 7 8] [9]
```

5. We want to shuffle the sequences to make it more realistic:

```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
dataset = dataset.map(lambda window: (window[:-1],
window[-1:]))
dataset = dataset.shuffle(buffer_size=10)
for x,y in dataset:
 print(x.numpy(), y.numpy())
 print("y = ", y.numpy())
[3 4 5 6] [7]
[5 6 7 8] [9]
[2 3 4 5] [6]
[4 5 6 7] [8]
[0 1 2 3] [4]
[1 2 3 4] [5]
```

6. We will create batches as follows:

```
dataset = dataset.shuffle(buffer_size=10)
dataset = dataset.batch(2).prefetch(1)
for x,y in dataset:
 print("x = ", x.numpy())
x = [[0 1 2 3]
[1 2 3 4]]
y = [[4]
[5]]
x = [[4 5 6 7]
[2 3 4 5]]
y = [[8]
[6]]
x = [[3 4 5 6]
[5 6 7 8]]
y = [[7]
[9]]
```

7. Next, we will learn these techniques to create a synthetic dataset.

Synthetic dataset

Let us create a dataset with seasonality, noise, and gradient that can be used for our analysis:

1. Let us first create plain linear series:

```
time = np.arange(4 * 365 + 1, dtype="float32")
baseline = 10
```

```
series = trend(time, 0.1)
series
```

2. Then, plot to help us visualize:

```
def plot_series(time, series, format="-", start=0,
end=None):
 plt.plot(time[start:end], series[start:end], format)
 plt.xlabel("Time")
 plt.ylabel("Value")
 plt.grid(True)
```

Figure 6.3 shows a basic series plot output:

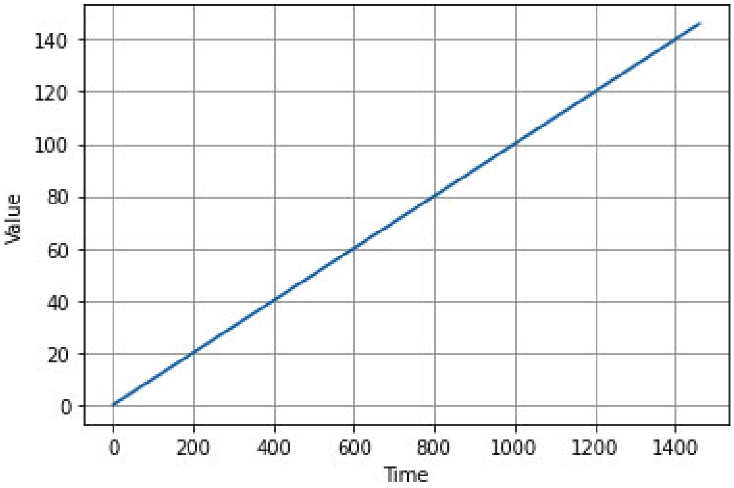


Figure 6.3: *Basic series plot generated*

3. Add **`baseline`** and **`seasonality`**:

```
amplitude = 50
slope = 0.05
noise_level = 10
def trend(time, slope=0):
  return slope * time
def seasonal_pattern(season_time):
  return np.where(season_time < 0.4,
np.sin(season_time * 2),
      np.cos(season_time * 2 * np.pi),
      1 / np.exp(3 * season_time))
def seasonality(time, period, amplitude=1, phase=0):
  season_time = ((time + phase) % period) / period
  return amplitude * seasonal_pattern(season_time)
def noise(time, noise_level=1, seed=None):
    rnd = np.random.RandomState(seed)
    return rnd.randn(len(time)) * noise_level
series = baseline + trend(time, slope) + seasonality(time,
period=365, amplitude=amplitude)
plot_series(time,series)
```

Plotting the series will give you a better idea. Note that the seasonal pattern is a random function, which has been applied a couple of times. Plot with the seasonality added is shown in *figure 6.4*:

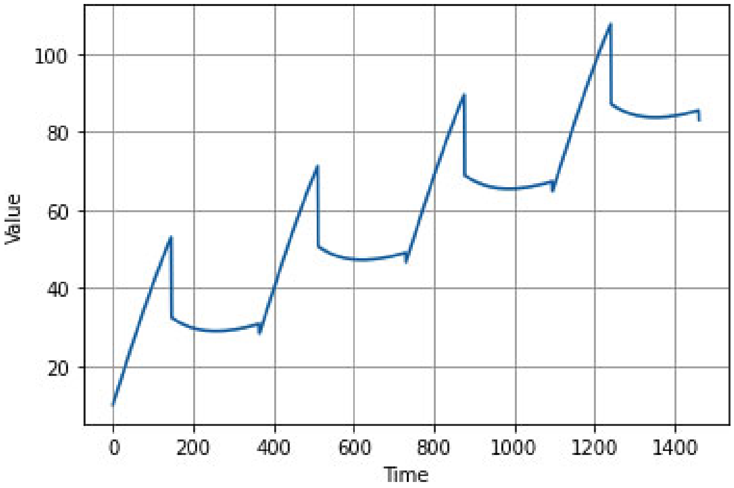


Figure 6.4: *Series plot with trend and seasonality*

4. Let us now apply some noise to the series, and plot it again:

```
series += noise(time, noise_level, seed=42)
plot_series(time,series)
```

Plot with noise added to the series can be seen in *figure 6.5*:

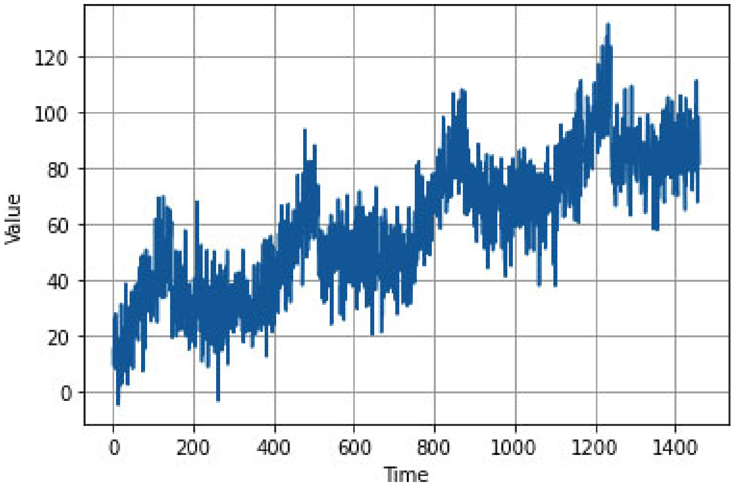


Figure 6.5: *Series plot with trend, seasonality and noise*

5. Create a split in the dataset for training and validation:

```
split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]
window_size = 20
batch_size = 32
shuffle_buffer_size = 1000
```

6. Create a **`windowed`** dataset:

```
def windowed_dataset(series, window_size, batch_size,
shuffle_buffer):
 dataset = tf.data.Dataset.from_tensor_slices(series)
 dataset = dataset.window(window_size + 1, shift=1,
 drop_remainder=True)
 dataset = dataset.flat_map(lambda window: window.batch(
window_size + 1))
 dataset = dataset.shuffle(shuffle_buffer).map(lambda
 window: (window[:-1], window[-1]))
 dataset = dataset.batch(batch_size).prefetch(1)
 return dataset
dataset = windowed_dataset(x_train, window_size, batch_size,
shuffle_buffer_size)
print(dataset)
```

Single layer model to predict time series

Follow these steps:

1. Let us create a single layer Keras model:

```
layer1 = tf.keras.layers.Dense(1, input_shape=[window_size])
model = tf.keras.models.Sequential([layer1])
model.compile(loss="mse",
optimizer=tf.keras.optimizers.SGD(lr=1e-6, momentum=0.9))
model.fit(dataset,epochs=100,verbose=0)
print("Layer weights {}".format(l0.get_weights()))
```

Let us create a **forecast** and print against the original values:

```
forecast = []
for time in range(len(series) - window_size):
  forecast.append(model.predict(
    series[time:time + window_size][np.newaxis]))
forecast = forecast[split_time-window_size:]
results = np.array(forecast)[:, 0, 0]
plt.figure(figsize=(10, 6))
plot_series(time_valid, x_valid)
plot_series(time_valid, results)
```

2. Let us plot the forecasted results and the actual ones, refer to *figure 6.6*. Notice how close the predictions are:

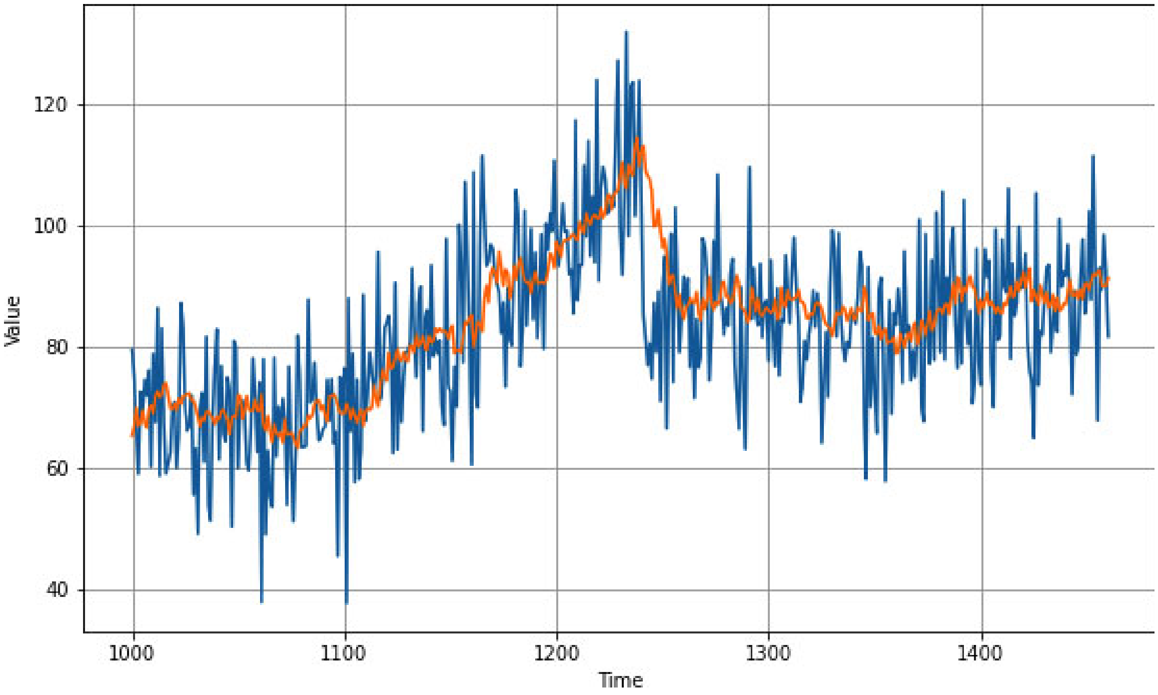


Figure 6.6: *Single layer DNN – predicted versus actual values*

3. Next, let us calculate some of the metrics to determine the effectiveness of the model:

```
tf.keras.metrics.mean_absolute_error(x_valid,
results).numpy()
```

The value of MAE after executing the preceding statement will be about `8.558071`.

Multiple layer model for predicting the value

Follow these steps:

1. Next, let us modify the model and include multiple layers:

```
dataset = windowed_dataset(x_train, window_size, batch_size,
shuffle_buffer_size)
print(dataset)
layer1 = tf.keras.layers.Dense(10, input_shape=
[window_size],activation="relu")
layer2 = tf.keras.layers.Dense(10, activation="relu")
output_layer = tf.keras.layers.Dense(1)
model = tf.keras.models.Sequential([layer1,layer2,
output_layer])
model.compile(loss="mse",
optimizer=tf.keras.optimizers.SGD(lr=1e-6, momentum=0.9))
model.fit(dataset,epochs=100,verbose=0)
```

2. With this model, the predicted result is similar to the following output:

```
forecast = []
for time in range(len(series) - window_size):
 forecast.append(model.predict(series[time:time +
 window_size][np.newaxis]))
forecast = forecast[split_time-window_size:]
results = np.array(forecast)[:, 0, 0]
plt.figure(figsize=(10, 6))
plot_series(time_valid, x_valid)
plot_series(time_valid, results)
```

3. The output is very similar to the single layer above, albeit with lower MAE. *Figure 6.7* shows the output of multi-layer DNN and the predicted line:

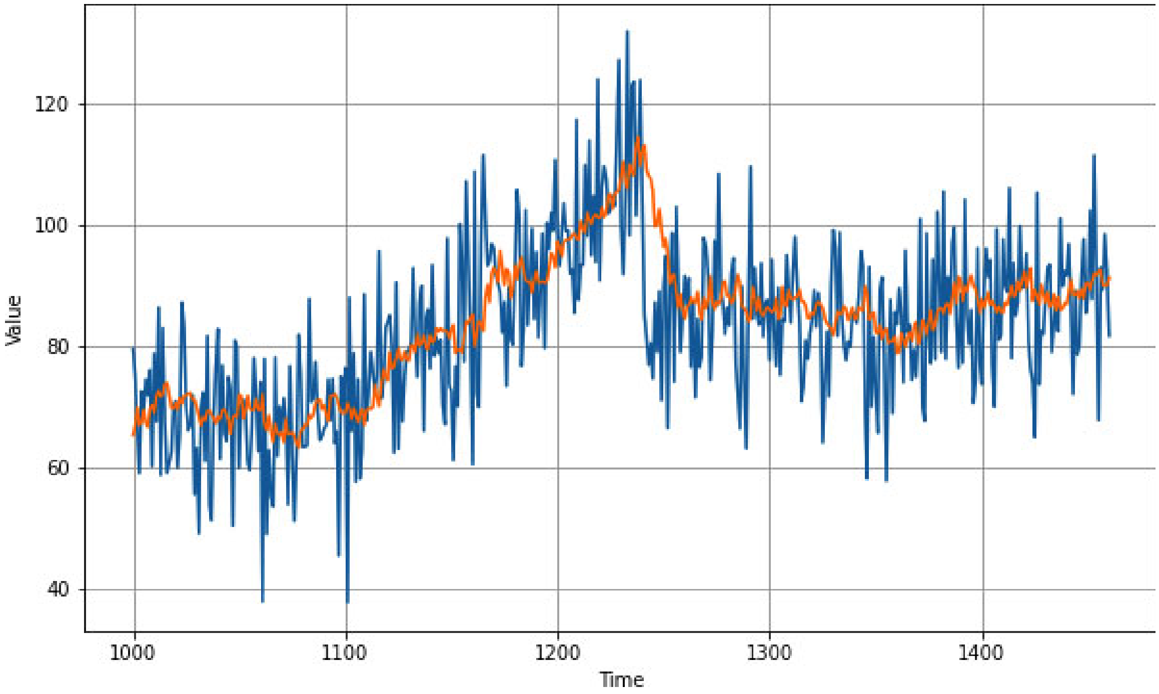


Figure 6.7: *Multi-layer DNN – predicted versus actual values*

4. On calculating the MAE loss using **`tf.keras.metrics.mean_absolute_error(x_valid, results).numpy()`**, we got a value of **`8.635604`**.
5. Let us also calculate the loss as a function of learning rate. We will be using the learning rate scheduler:

```
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(
  lambda epoch: 1e-8 * 10**(epoch / 20))
model.compile(loss="mse",
optimizer=tf.keras.optimizers.SGD(lr=1e-6,
  momentum=0.9))
history = model.fit(dataset,epochs=100,
         verbose=0,
              callbacks=[lr_scheduler])
```

6. Once we have the history of loss values for each epoch, let us plot them on a **`semilogx`** scale:

```
lrs = 1e-8 * (10 ** (np.arange(100) / 20))
plt.semilogx(lrs, history.history["loss"])
plt.axis([1e-8, 1e-3, 0, 300])
```

[*Figure 6.8*]() is the output which shows that the loss remains almost flat, and after **`1e`**$^{-5}$ climbs up again:

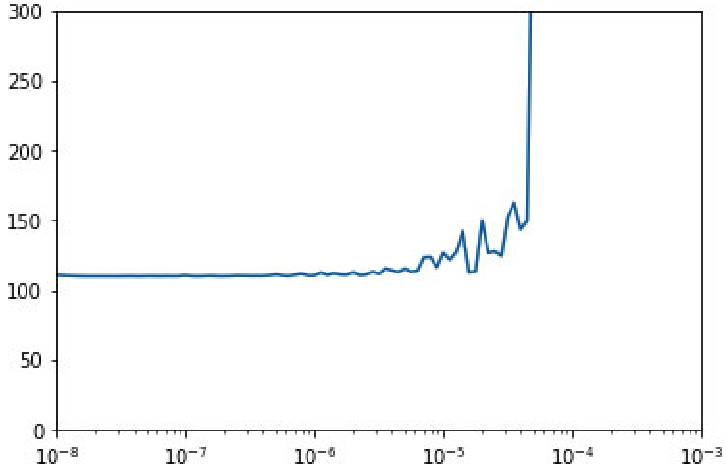


Figure 6.8: *Multi-layer DNN – predicted versus actual values*

7. Let us change the learning rate to **`1e-5`**, and see the output:

```
model.compile(loss="mse",
optimizer=tf.keras.optimizers.SGD(lr= 1e-5, momentum=0.9))
history = model.fit(dataset,epochs=100,verbose=0)
```

   Plotting the history will show the graph:

```
loss = history.history['loss']
epochs = range(len(loss))
plt.plot(epochs,loss,'b')
```

   As you can see in *figure 6.9*, the loss remains more or less fluctuating in the same range without any improvement:

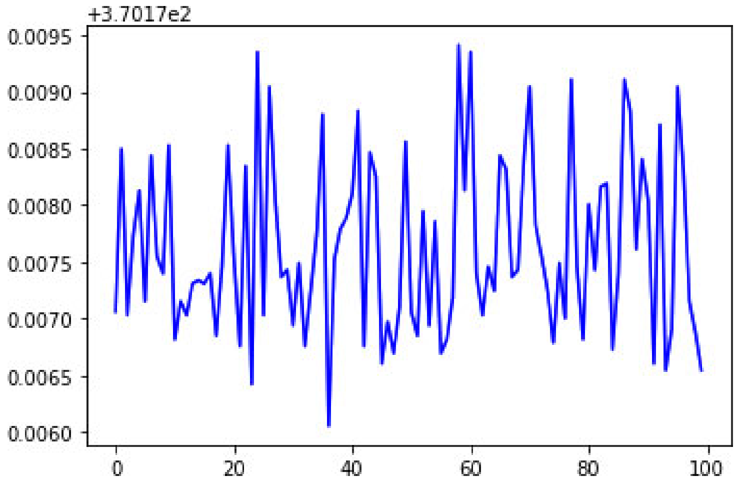


Figure 6.9: *Loss as a function of epochs for learning rate of 1e-5*

8. Let us look at **`loss`** as a function of epochs for various learning rates:

```
lr_array = [1e-4,1e-5,1e-6,8e-6,1e-7,1e-8]
history_array = []
```

```
for _lr in lr_array:
 model.compile(loss="mse", optimizer=tf.keras.optimizers.
 SGD(lr=_lr,
      momentum=0.9))
 _history = model.fit(dataset,epochs=100,verbose=0)
 _loss = _history.history['loss']
 history_array.append(_loss)
for i in range(0,5):
 plt.plot(epochs,history_array[i],label=lr_array[i])
plt.legend(bbox_to_anchor=(1.05, 1))
plt.title('Loss vs Epochs for various Learning Rates')
```

Figure 6.10 shows how the loss varies as a function of various learning rates, it settles down and around `0.0765` for `1e-07` and `1e-08`:

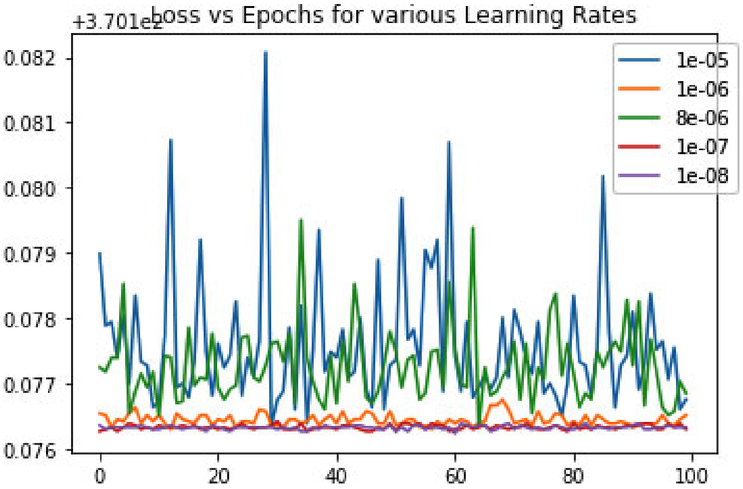


Figure 6.10: *Loss as a function of epochs for various learning rates*

In the next section, we will look at how to process the time series dataset using more sophisticated networks like **Recurrent Neural Networks** (**RNN**).

RNN for predicting the time series data

A **Recurrent Neural Network,** also known as **RNN,** is a neural network that contains recurrent layers. These are designed to sequentially process a sequence of inputs. RNNs are quite flexible, able to process all kinds of sequences. They can be used for predicting texts, as we saw earlier. In this section, we'll use them to process the time series dataset we created.

This example will build an RNN which contains two recurrent layers and a dense layer, which will serve as the output layer. An RNN can be fed batches of sequences, and the network will output a batch of forecasts. One difference from DNNs, that we saw earlier, is that the full input shape is three-dimensional. The first dimension is the **batch size**, the second is **timestamp**, and the third is the **dimensionality** of the inputs for each time step. In case of a univariate time series, this value will be one; for **multivariate**, it'll be more. *Figure 6.11* illustrates a simple three layered RNN:

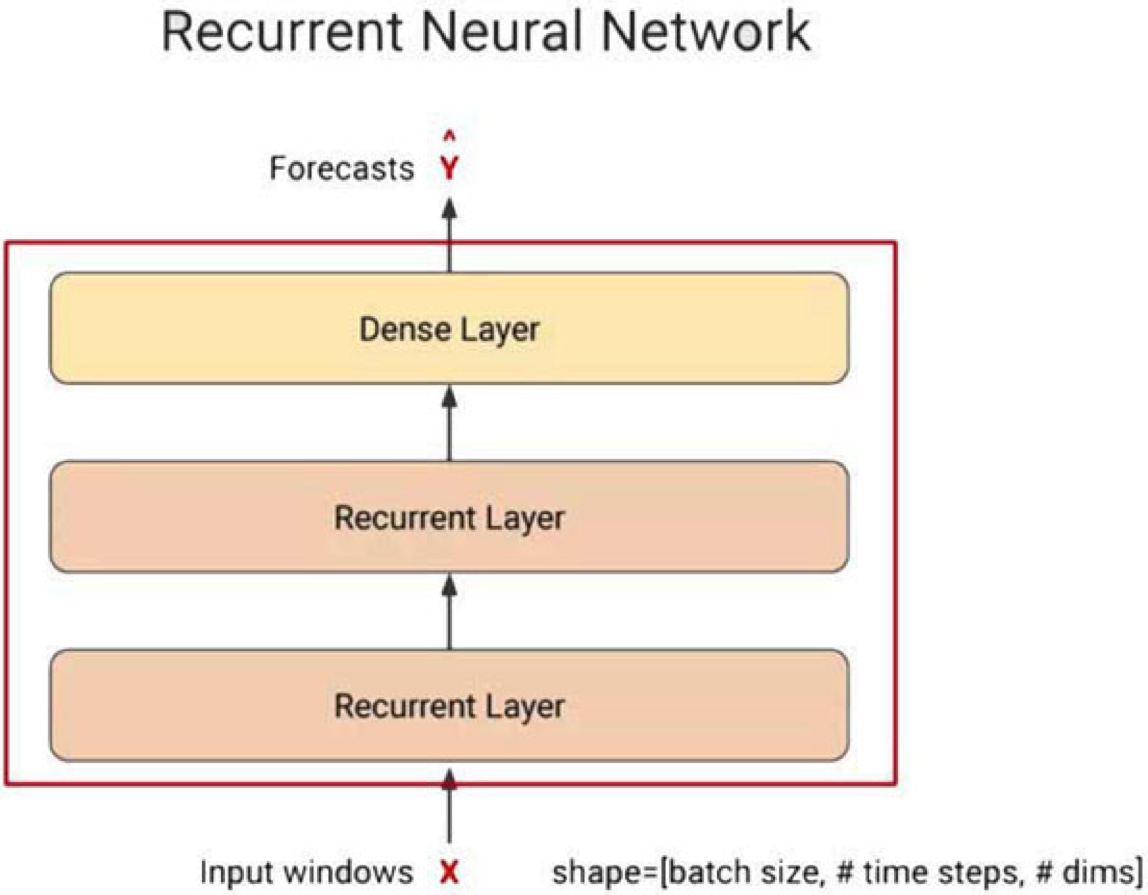


Figure 6.11: *Recurrent layers in a three layer RNN*

Figure 6.12 illustrates how multiple recurrent layers form an RNN:

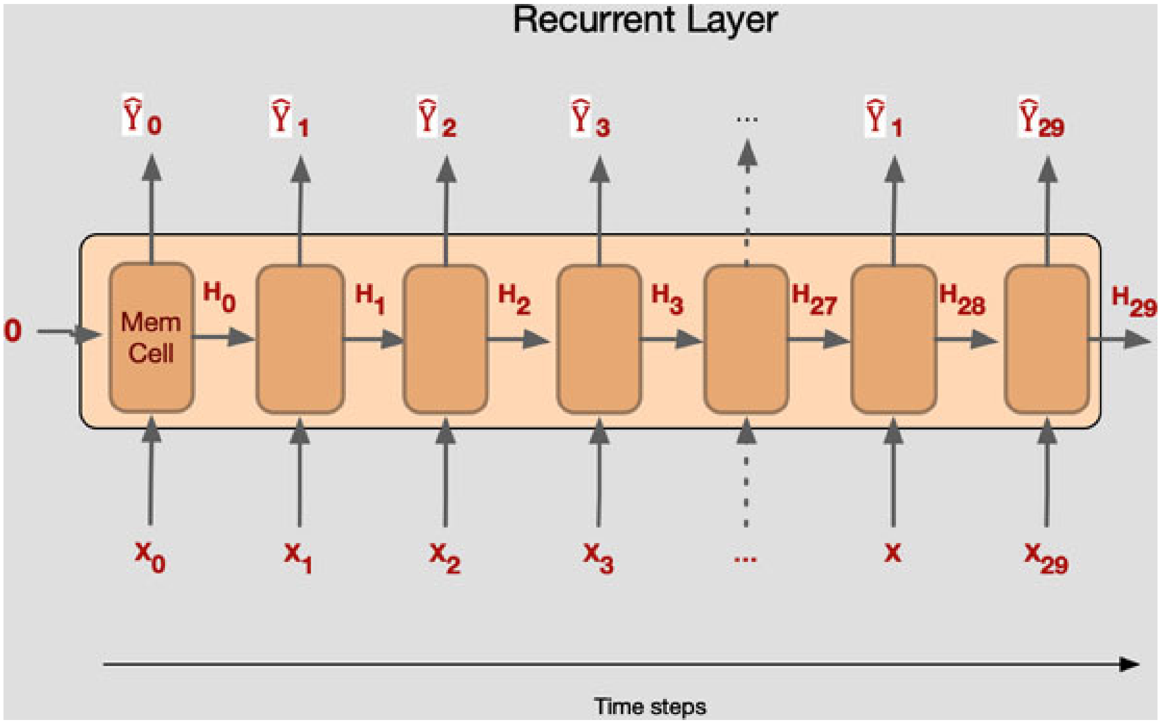


Figure 6.12: *Multiple recurrent layers in an RNN*

Even though it looks like it has a lot of cells, it is actually one cell that is used repeatedly. At each step, the input is fed into the cell. There are two outputs that will come out the state output and the actual output .

The RNN network, as shown, starts with **0** state, input of, and output of first cell and state of . This continues for each time step, till we reach the end of the time step.

The inputs which go into an RNN are three dimensional. For example, if we have a window size of *30 timestamps* and we create a batch of four, the shape will be *4 x 30 x1*, and each timestamp, the memory cell input will be a four by one matrix.

The cell will also take the input of the state matrix from the previous step. In the first step, the state matrix will be zero. For subsequent ones, it will be the output from the memory cell. But other than the state vector, the cell will output a *Y* value. *Figure 6.13* illustrates the data in batch mode, where each time is step being fed into recurrent layers in a RNN:

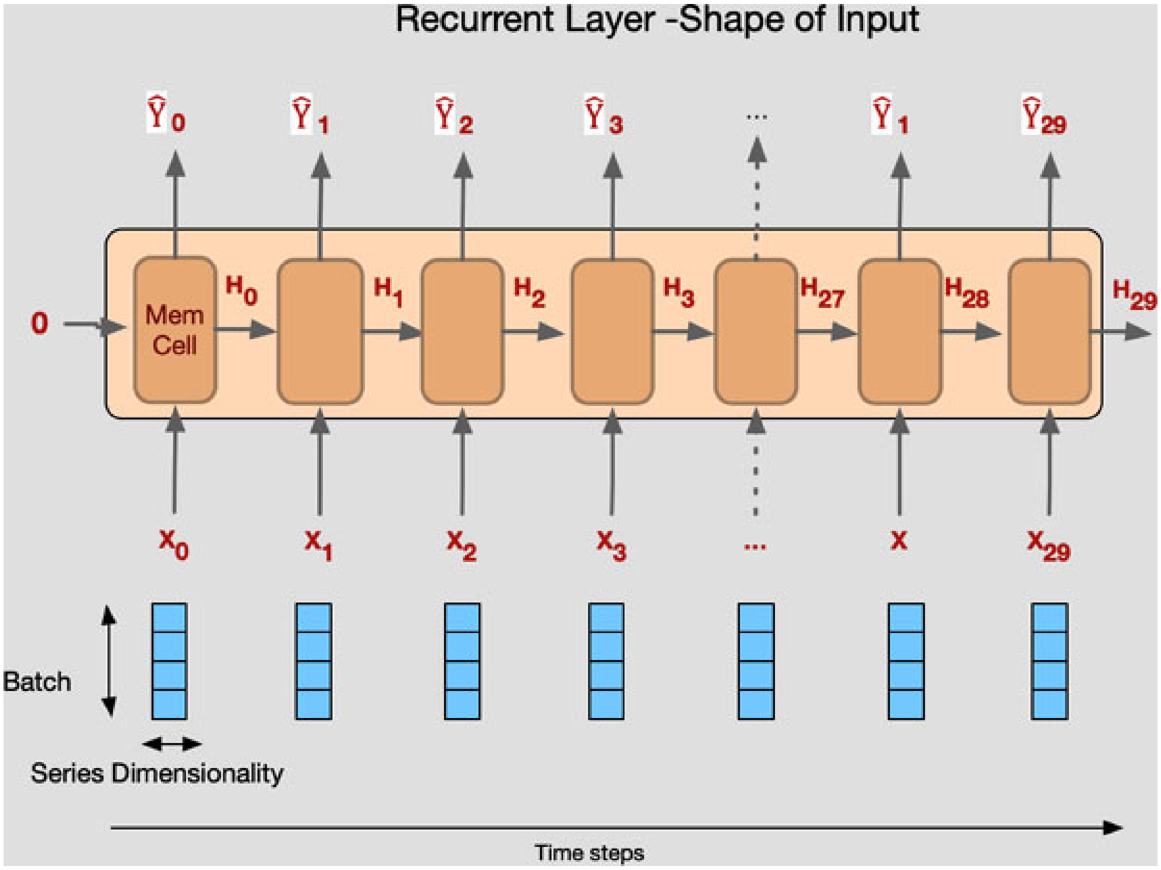


Figure 6.13: *Multiple recurrent layers in an RNN*

In the next section, we will look at how RNN can be used for time series prediction of synthetic dataset.

Return sequences in RNN, and application to time series

Simple RNN layers in TensorFlow can either return a single output or sequence of outputs depending on the parameter **`return_sequence`**:

```
tf.keras.layers.SimpleRNN(40, return_sequences=True),
tf.keras.layers.SimpleRNN(40),
```

In the first line of code, the layer's output will be a sequence as shown in the following *figure 6.14*:

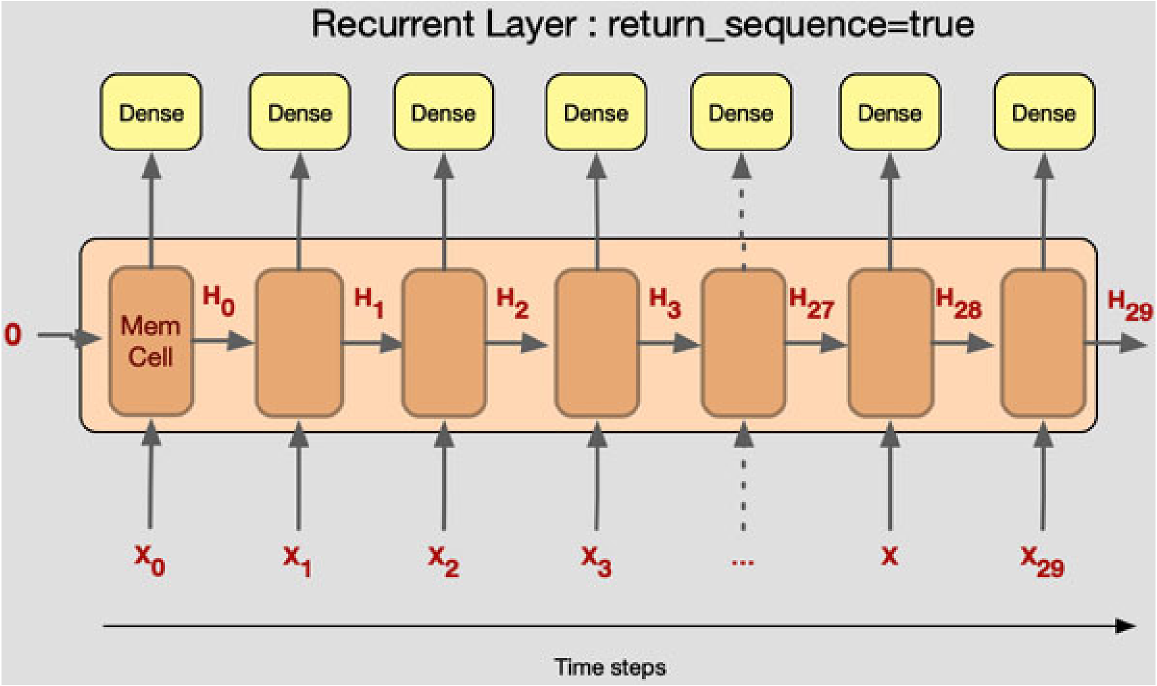


Figure 6.14: *RNN layer with multiple outputs going into same dense layer at each time stamp*

If parameter **`input_sequence=false`**, there will be only one output coming out of the final step in the RNN network.

Keras handles this feature by using the same dense layer at each time stamp.

It might look like multiple dense layers, but it's the same one that is being reused at each time step. This gives the topology called a **sequence-to-sequence RNN**. It's fed a batch of sequences, and it returns a batch of sequences of the same length:

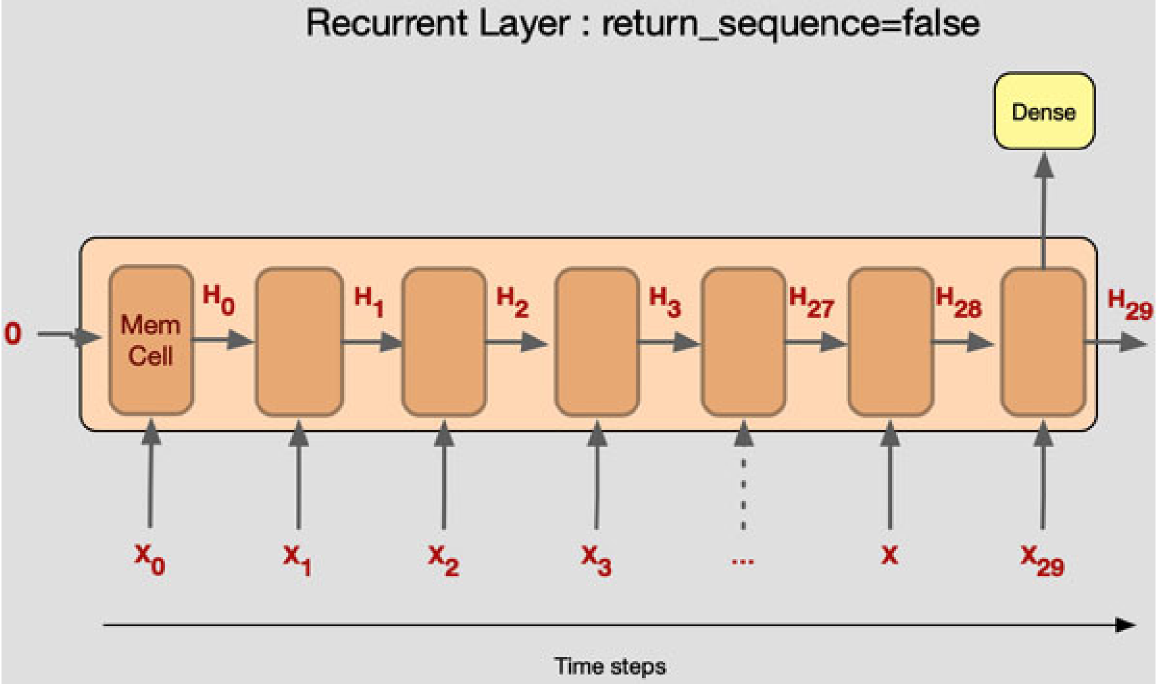


Figure: 6.15: *Multi-layer feedforward network on the generated dataset: RNN layer which has a single output*

Figure 6.15 is an example of a recurrent layer being fed the output from the previous layer only. There is only one output coming out of the RNN, and going into the dense layer.

Applying Lambda function

We also want to learn how to apply *Lambda function* to the output of a RNN layer. This will help apply simple RNN to the synthetic dataset. When we implemented the `window` dataset helper function earlier, it returned two-dimensional batches of windows on the data. First parameter being the **batch size,** and the second, the **number of timestamps**.

An RNN, on the other hand, expects three-dimensions: **batch size**, the **number of timestamps**, and the **series dimensionality**.

This can be fixed without rewriting our `window` dataset helper function by using Lambda layer.

We expand the array by one dimension using the `lambda` function. Setting input shape to `None`, means that the model can take sequences of any length.

We can help training by scaling up the outputs by `100`. The default activation function of the RNN layers is `tanh` – this is a hyperbolic tangent activation.

This function outputs values between negative one and one:

```
tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
  input_shape=[None]),
  tf.keras.layers.SimpleRNN(40, return_sequences=True),
  tf.keras.layers.SimpleRNN(40),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 100.0)
```

The preceding code listing shows how the Lambda functions can be used as the first layer, and the last layer.

Adjusting the learning rate dynamically

Instead of having a fixed learning rate, we can have a dynamic learning rate that is controlled by a learning rate scheduler:

```
optimizer = tf.keras.optimizers.SGD(lr=5e-5, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
  optimizer=optimizer,
  metrics=["mae"])
```

We are using a different loss function called **Huber**, it is less sensitive to outliers. In the following example, it varies between **`1e-8`** and **`1e-4`**. The following figure shows how the loss is minimized for a learning rate of about **`1e-5`**:

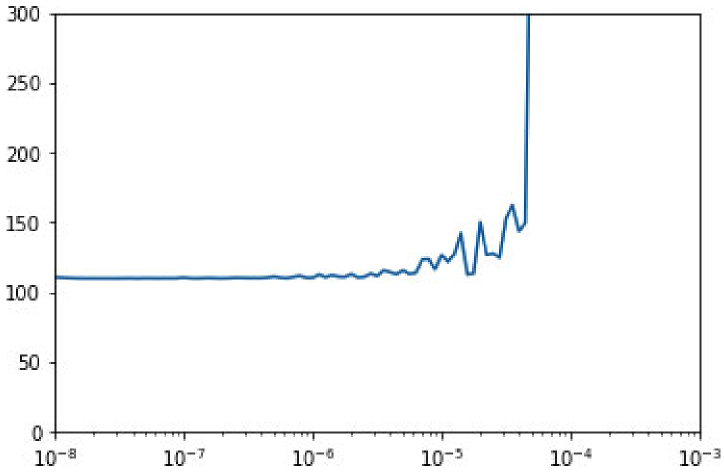


Figure 6.16: *Loss versus learning rate for RNN*

Putting it all together, and compiling the model leads to the following output:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
 input_shape=[None]),
 tf.keras.layers.SimpleRNN(40, return_sequences=True),
 tf.keras.layers.SimpleRNN(40),
 tf.keras.layers.Dense(1),
 tf.keras.layers.Lambda(lambda x: x * 100.0)
])
optimizer = tf.keras.optimizers.SGD(lr=5e-5, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
 optimizer=optimizer,
 metrics=["mae"])
history = model.fit(dataset,epochs=400)
Let us plot the actual value versus prediction:
forecast=[]
for time in range(len(series) - window_size):
 forecast.append(model.predict(series[time:time + window_size]
 [np.newaxis]))
forecast = forecast[split_time-window_size:]
results = np.array(forecast)[:, 0, 0]
plt.figure(figsize=(10, 6))
plt.title('Actual vs Predicted value for SimpleRNN Network')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plot_series(time_valid, x_valid)
plot_series(time_valid, results)
```

Output of the plot is shown in the following figure:

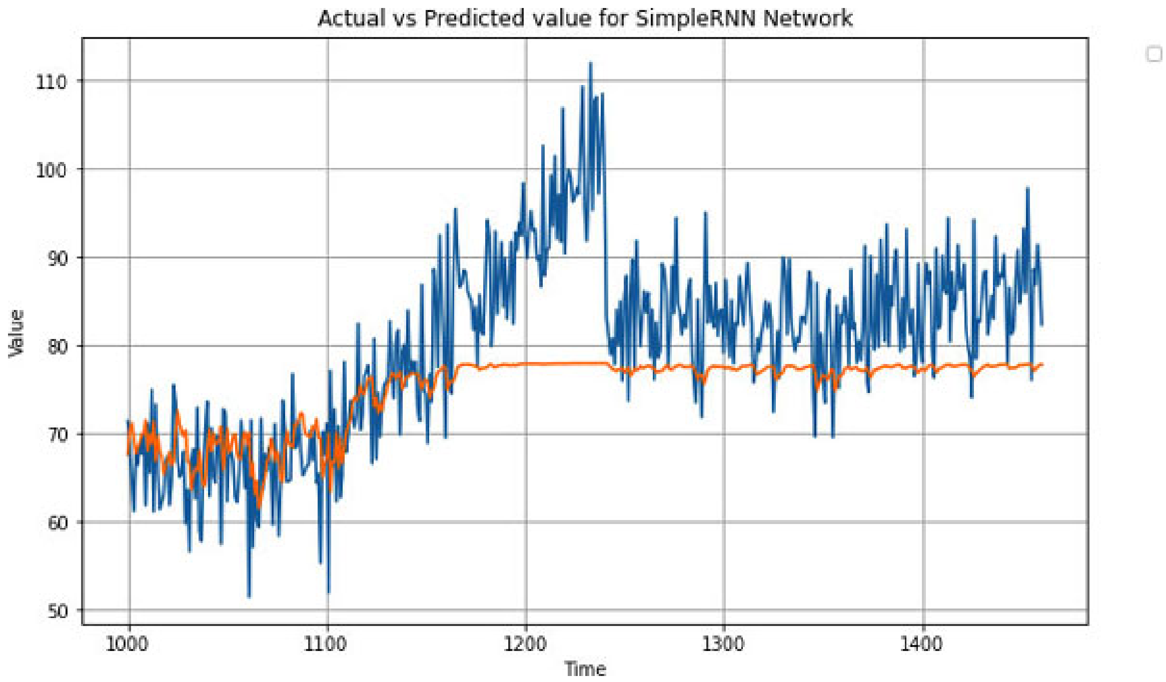


Figure 6.17: *Prediction versus actual for RNN*

We will also calculate the value of MAE using `tf.keras.metrics.mean_absolute_error(x_valid, results).numpy()`.

We got a value of `7.3184185`, which is much lower than a single DNN `9.342689`.

Plot the loss

1. Let us plot the `mae` and `loss` as a function of epochs:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
mae=history.history['mae']
loss=history.history['loss']
epochs=range(len(loss)) # Get number of epochs
#------------------------------------------------
# Plot MAE and Loss
#------------------------------------------------
plt.plot(epochs, mae, 'r')
plt.plot(epochs, loss, 'b')
plt.title('MAE and Loss')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["MAE", "Loss"])
plt.figure()
epochs_zoom = epochs[200:]
mae_zoom = mae[200:]
loss_zoom = loss[200:]
#-------------------------------------------------
# Plot Zoomed MAE and Loss
#-------------------------------------------------
plt.plot(epochs_zoom, mae_zoom, 'r')
plt.plot(epochs_zoom, loss_zoom, 'b')
plt.title('MAE and Loss')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend(["MAE", "Loss"])
plt.figure()
```

Output of the preceding code in *figure 6.18* shows how the loss minimizes by about **400 epochs**:

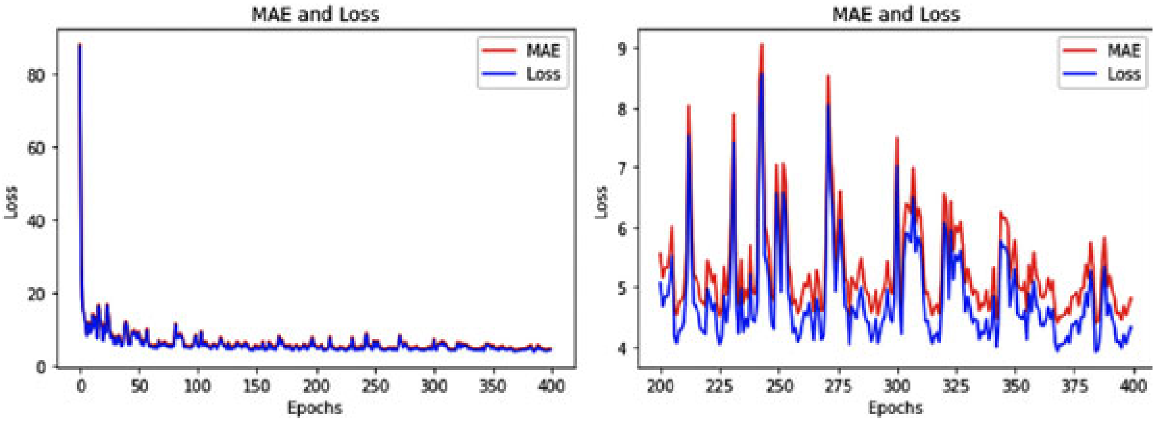


Figure 6.18: *Loss for simple RNN as a function of epochs, right plot shows plot from 200 to 400 epochs*

We can also see that MAE and loss is on its way of converging to **6.6**. In the next section, we look at how LSTMs can be used in handling the time series data.

LSTM and time series

We have learnt that RNN provides improvements over deep neural fully connected networks. We learnt that RNN processes the input from subsequent time stamps in a batch, and gives output Y_t and cell state H_t. Figure 6.19 illustrates how state H_t is copied:

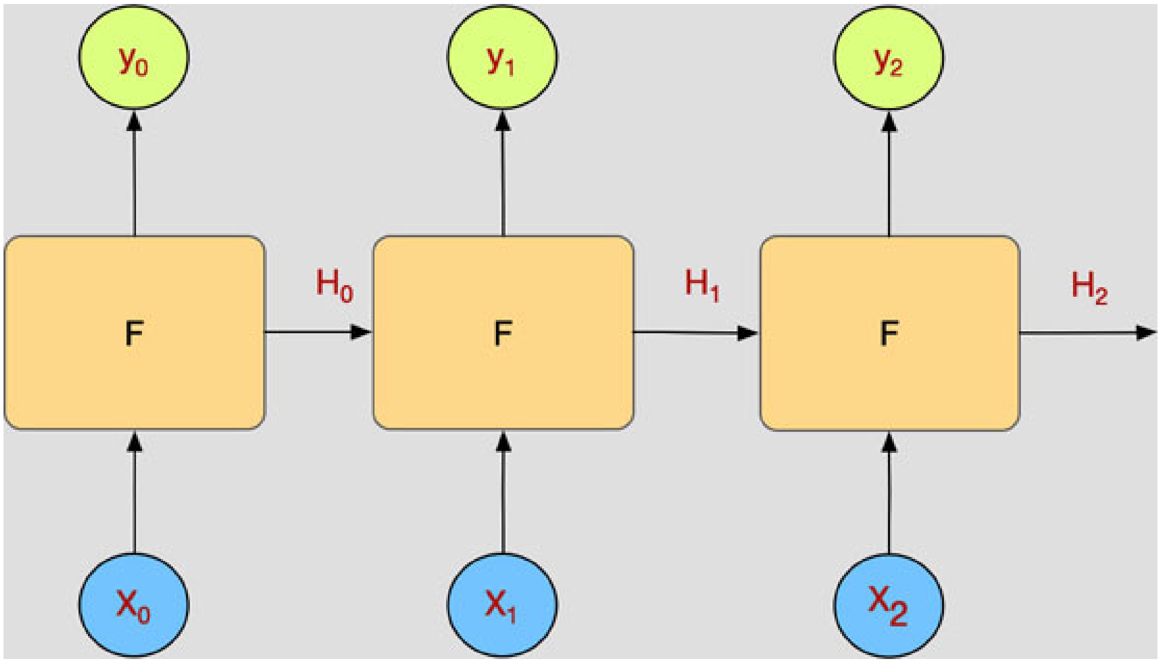


Figure 6.19: *RNN network with cell state H copied from one cell to another*

While the cell state is carried forward, its effect is diminished. In **LSTM**, cell state is carried over right through the cells. This state can move in one direction, where the LSTM is called **unidirectional** as shown in *figure 6.20*:

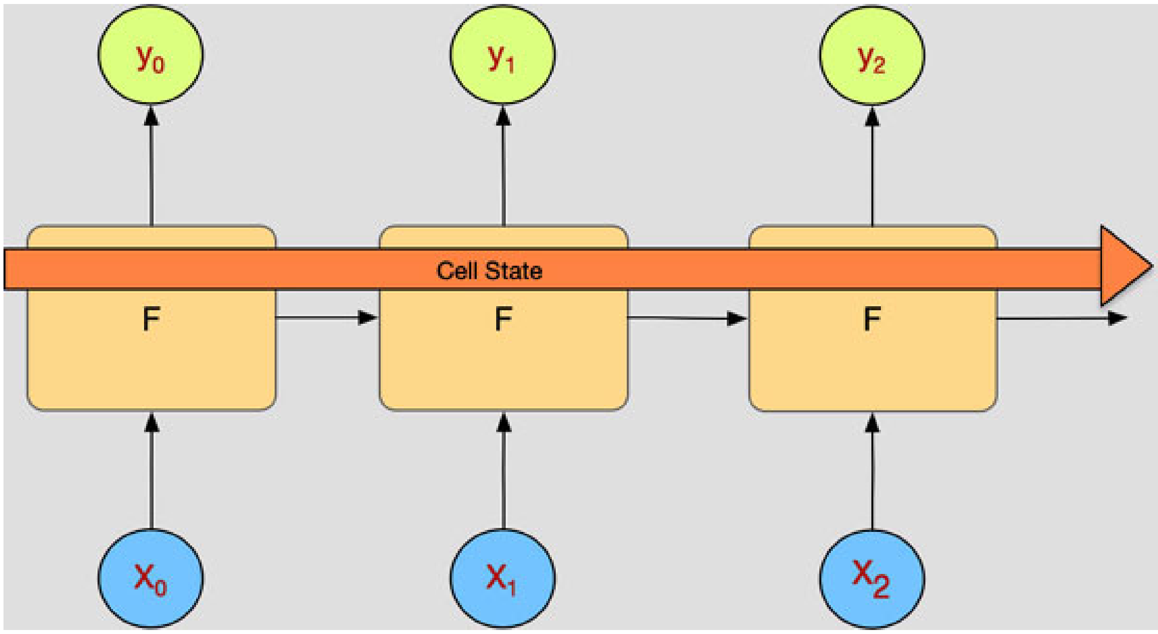


Figure 6.20: *Unidirectional LSTM network*

In case of **bidirectional LSTM network**, it moves forward as well as backwards between cells (refer to *figure 6.21*):

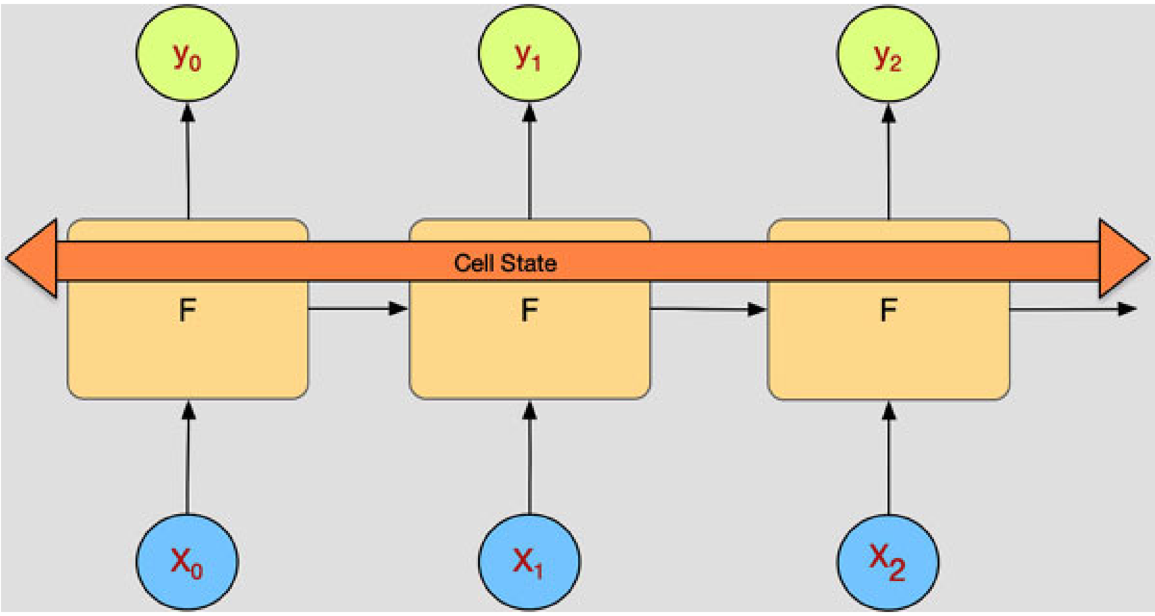


Figure 6.21: *Bidirectional LSTM network*

Process synthetic dataset with LSTM

Let us apply LSTM to the synthetic dataset we created earlier. TensorFlow comes with an inbuilt implementation of LSTM layer, which we are planning to use:

```
from synthetic_dataset_util import get_series
from synthetic_dataset_util import get_x_train_x_valid
series = get_series()
split_time = 1000
x_train, x_valid = get_x_train_x_valid()
window_size = 20
batch_size = 32
shuffle_buffer_size = 1000
tf.keras.backend.clear_session()
tf.random.set_seed(51)
np.random.seed(51)
from synthetic_dataset_util import windowed_dataset
tf.keras.backend.clear_session()
dataset = windowed_dataset(x_train, window_size, batch_size,
shuffle_buffer_size)
Let us create our LSTM based sequence model:
model = tf.keras.models.Sequential([
  tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
        input_shape=[None]),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32,
            return_sequences=True)),
  tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 100.0)])
```

We have bidirectional LSTM layers with **32 cells**. The first one returns sequences, while the second one gives a single output from the last layer:

```
lr_schedule = tf.keras.callbacks.LearningRateScheduler(
    lambda epoch: 1e-8 * 10**(epoch / 20))
optimizer = tf.keras.optimizers.SGD(lr=1e-8, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
        optimizer=optimizer,
        metrics=["mae"])
history = model.fit(dataset, epochs=100, callbacks=[lr_schedule])
```

Let us plot the loss for various learning rates:

```
plt.semilogx(history.history["lr"], history.history["loss"])
plt.axis([1e-8, 1e-4, 0, 30])
```

We can see that the loss minimizes around `1e-5`. *Figure 6.22* shows the plot of loss versus learning rate for RNN:

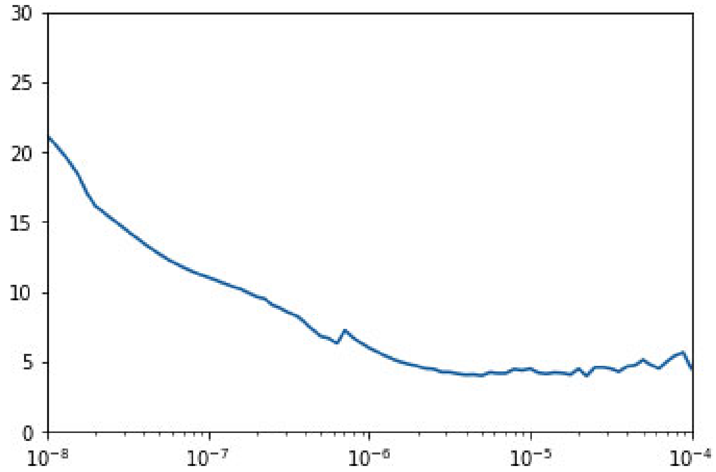


Figure 6.22: *Semilogx plot for loss versus learning rate for RNN*

For this network, the plot of forecast versus actual looks very similar with a lower MAE:

```
forecast = []
results = []
for time in range(len(series) - window_size):
  forecast.append(model.predict(series[time:time + window_size]
  [np.newaxis]))
forecast = forecast[split_time-window_size:]
results = np.array(forecast)[:, 0, 0]
plt.figure(figsize=(10, 6))
plot_series(time_valid, x_valid)
plot_series(time_valid, results)
```

The output of the two plots is as shown in the following figure:

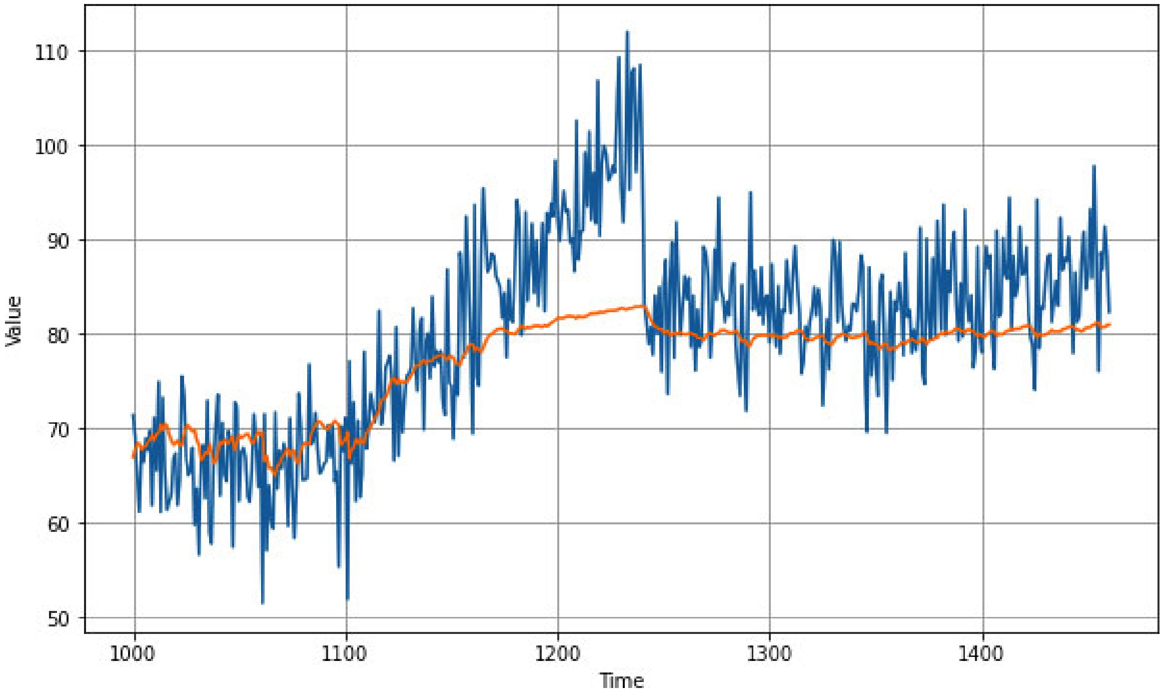


Figure 6.23: *Real versus predicted plot for the dataset*

Next, let us look at the MAE: **`tf.keras.metrics.mean_absolute_error(x_valid, results).numpy()`**. This is coming to around **`5.8380394`**. We will also plot MAE and loss as a function of iterations:

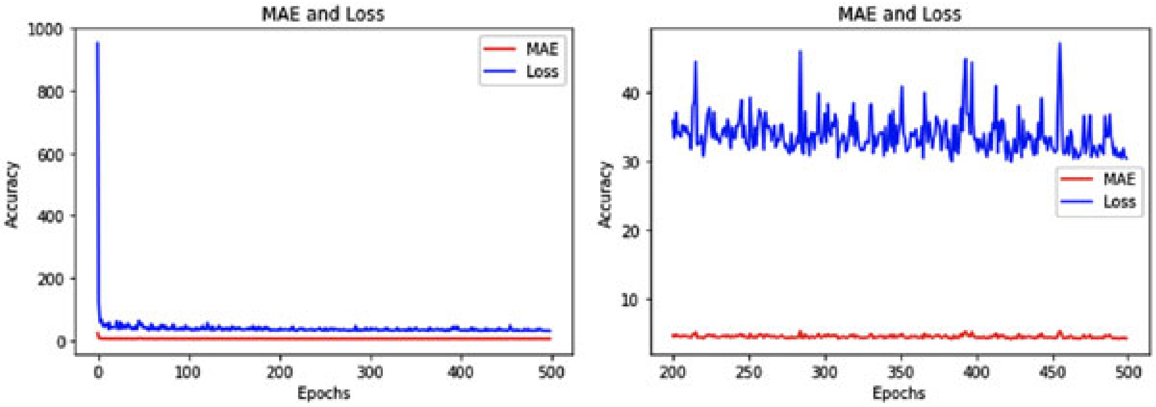


Figure 6.24: *MAE and loss as a function of epochs for bidirectional LSTM*

With this analysis, we can conclude that LSTM provides the most state or the art accuracy for this simple time series synthetic data we analyzed.

Conclusion

In this chapter, we learnt how to use various techniques in TensorFlow, like RNN and LSTM, to predict values for a time series dataset. We learnt how to leverage `tf.dataset` to create a `windowed` dataset, and processed it with various topologies and compared the results.

References

- **Time series**: https://en.wikipedia.org/wiki/Time_series.
- **tf.keras.layers.LSTM**: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM.
- **Recurrent Neural Networks (RNN) with Keras**: https://www.tensorflow.org/guide/keras/rnn.
- **Sequences, Time Series and Prediction**: https://www.coursera.org/learn/tensorflow-sequences-time-series-and-prediction/home/welcome.

Points to remember

- A time series data needs a special kind of network which can retain state.
- RNN and LSTM are such networks that maintain state across timestamps.

Multiple choice questions

1. **LSTM is the most effective to predict time series:**

   a. true

   b. false

2. **Simple RNN always returns sequences:**

   a. true

   b. false

3. **If X is the standard notation for the input to an RNN, what are the standard notations for the outputs?**

   a. $\hat{Y}$

   b. H

c. $\hat{Y}$ and H

4. **The function of a Lambda layer in a neural network is:**

    a. to execute arbitrary code during training

    b. to change the shape of the input

    c. to allow pausing the operation

    d. none of the above

5. **How do you clear temporary variables created in a TensorFlow session?**

    a. tf.clear_session()

    b. tf.keras.clear_session()

    c. tf.cache.clear_session()

Answers

1. **b**
2. **b**
3. **c**
4. **a**
5. **b**

Questions

1. What are the various techniques to handle a time series data in TensorFlow?
2. Does adding more RNN layers increase the accuracy of the output for the time series data?
3. What is a windowed dataset?
4. What is a bidirectional layer?

Key terms

- **RNN**: Recurrent Neural Networks
- **LSTM**: Long-Term Short-Term Memory

CHAPTER 7
Distributed Training

Introduction

In this chapter you will learn how TensorFlow supports multiple GPUs and CPUs on same or multiple devices to parallelize training. You will learn about the implementation of Mirrored Strategy as well as an example of distributed training with TensorFlow using MNIST dataset as an example. You will also learn about training with multiple workers using high level Keras based APIs.

Structure

In this chapter, will discuss the following topics:

- Overview of distributed training support in TensorFlow
- Implementation specifics
- How the implementation handles single and multi-device distributed training
- Code walk-through of the underlying implementation of **MirroredStrategy**
- Using distributed training for MNIST classification on AWS
- Multiple-worker training with Keras

Objectives

After studying this unit, you should be able to understand how TensorFlow distributed implementation works. You will learn about the various strategies like mirror strategy and TPU strategy available under the TensorFlow distributed. You will also understand the implementation details, and use a strategy with a well-known use case like MNIST classification.

Introduction to distributed TensorFlow

To overcome the limitation posed by training a large dataset on a single GPU or TPU, TensorFlow supports distributed training through standard APIs. There is a standard class `tf.distribute.Strategy` entry point for all distributed training implementations. It helps in distributing data across GPUs/TPUs for training. TPU stands for **Tensor Processing Unit**. It is an AI accelerator application specific integrated unit developed by Google for neural network machine learning use cases. It is available through Google Cloud: **Compute Engine**, **Kaggle**, and **Google Colab** based notebooks.

Types of strategies

There are two common patterns in distributed training—**sync** and **async**. In **sync training**, all workers train over different input data slides in sync and aggregate gradients at each step. In **async training**, all workers train asynchronously and update variables. Sync training is supported using all-reduce, and async with parameter server architecture.

There are six strategies available:

Training API	Mirrored strategy	TPU strategy	Multi-worker mirrored strategy	Central storage strategy	Parameter server strategy
Keras API	Supported	Supported	Experimental support	Experimental support	Supported in 2.3
Custom training loop	Supported	Supported	Experimental support	Experimental support	Supported in 2.3
Estimator API	Limited support	Not supported	Limited support		

***Table 7.1:** Six strategies and the support in various training APIs*

Let us go through each one of them in more detail in the next few sections.

Mirror strategy

Mirror strategy is the base strategy object to be used while distributing variable updates across GPUs, all within a single machine. Most of the cloud VMs have a limitation of *8 GPUs* per machine; the reference link is as follows:

https://cloud.google.com/compute/docs/gpus/

The following *figure 7.1* illustrates the basic building blocks of mirror strategy:

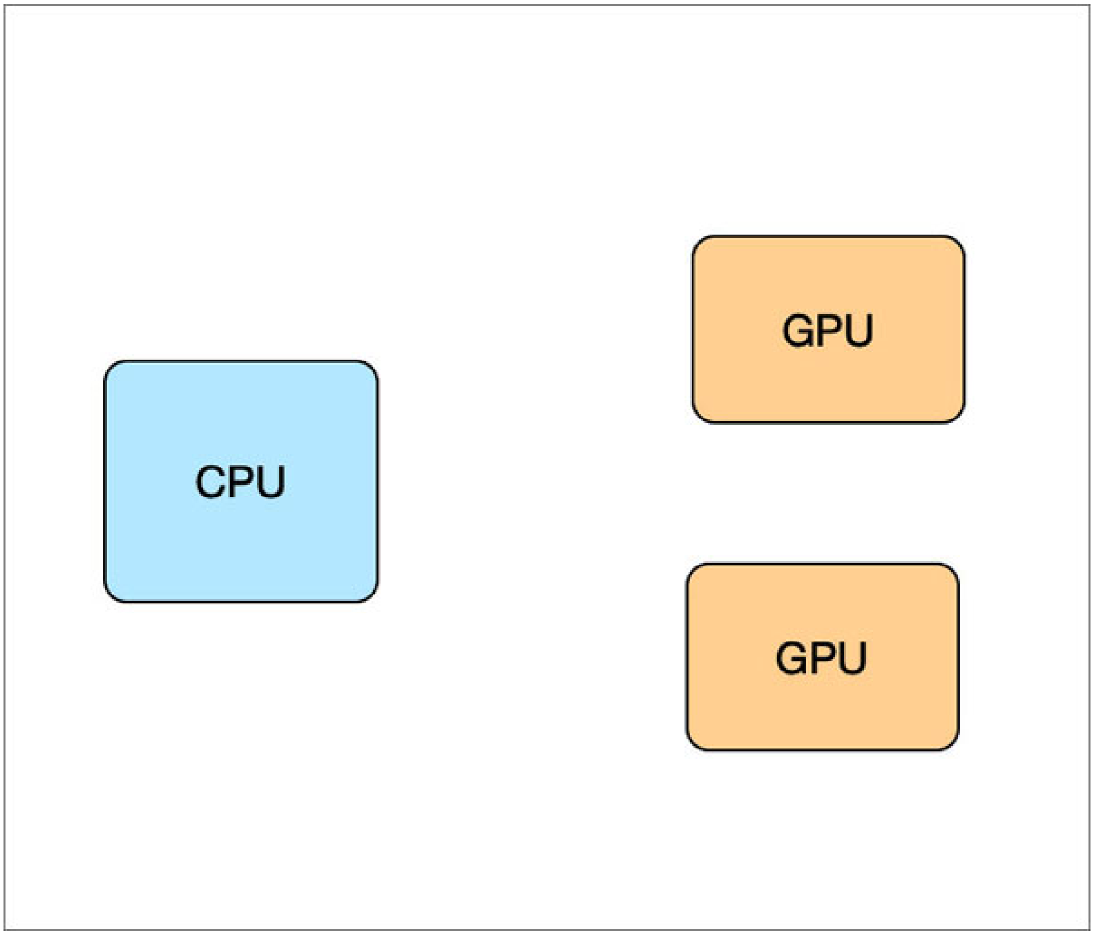


Figure 7.1: *Building blocks of mirror strategy*

Implemented in class **`tf.distribute.MirroredStrategy`**, it extends the base class **`tf.distribute.Strategy`**.

This strategy is typically used for training on one machine with multiple GPUs.

For TPUs, use **`tf.distribute.TPUStrategy`**. To use **`MirroredStrategy`** with multiple workers, we will look at **`tf.distribute.experimental.MultiWorkerMirroredStrategy`**.

For example, a variable created under a **`MirroredStrategy`** is a **`MirroredVariable`**. If no devices are specified in the constructor argument of the strategy, it will use all the available GPUs. If no GPUs are found, it will use the available CPUs. Note that TensorFlow treats all CPUs on a machine as a single device and uses threads internally for parallelism. The following code shows how to initialize the **`MirroredStrategy`**:

```
 mirrored_strategy = tf.dis tribute.MirroredStrategy()
```

Figure 7.2 illustrates the distribution strategy followed depending on the distribution strategy flag, number of workers, and number of GPUs available:

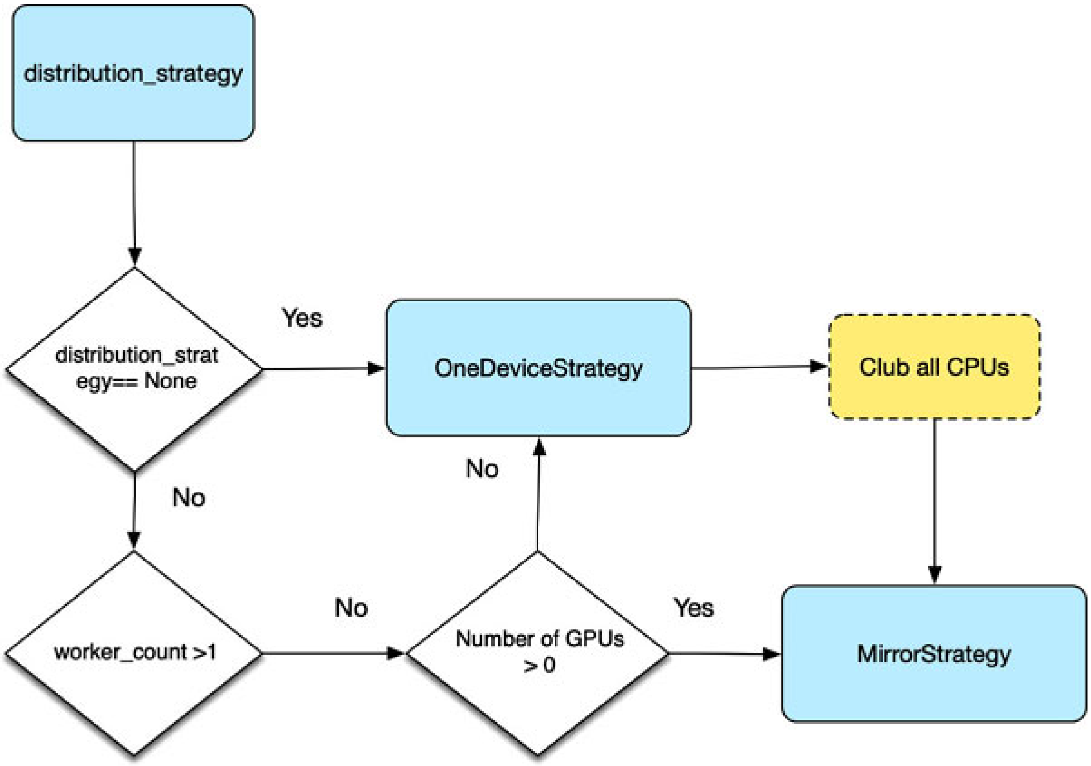


Figure 7.2: *Choosing the right strategy for distributed training*

In **MirroredStrategy**, there is one worker responsible for updating the **MirroredVariable** based on part of the dataset passed to that GPU.

This implementation of **MirroredStrategy** reduces to one GPU along edges in a hierarchy. Results are broadcast back to each GPU along the same path. Tensors are repacked or aggregated for more efficient cross-device transportation before performing all-reduce:

```
tf.distribute.HierarchicalCopyAllReduce(
    num_packs=1
)
```

Single-worker strategy with **MirroredVariable** is illustrated in the *figure 7.3*:

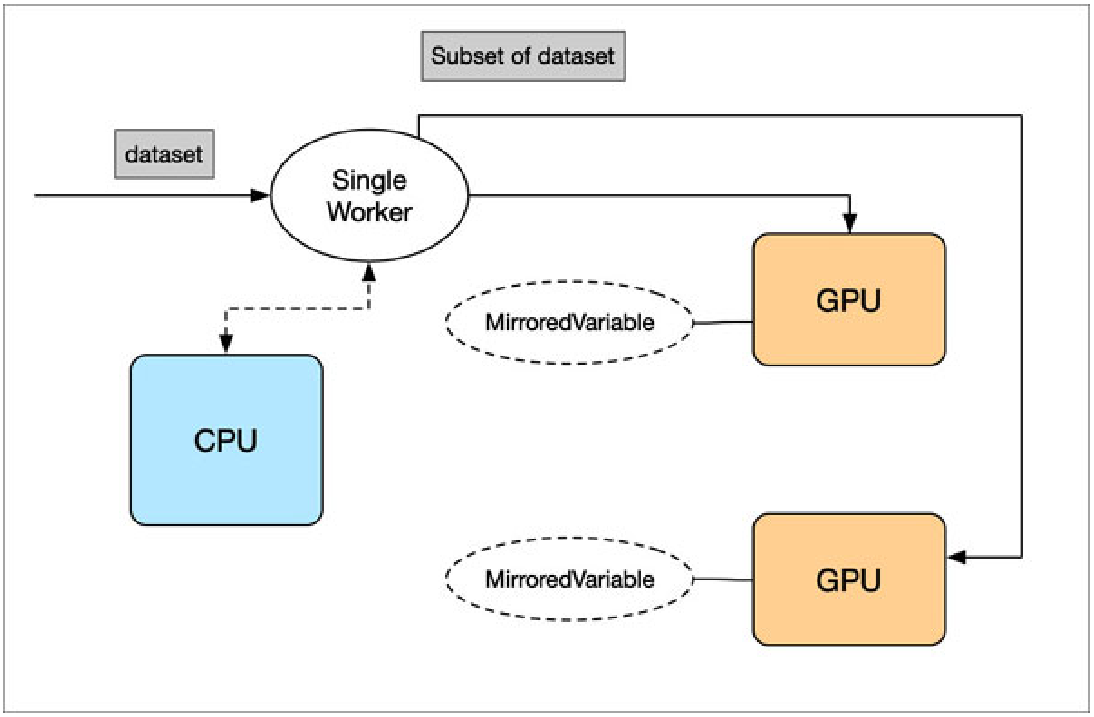


***Figure** 7.3: Single worker updating MirroredVariable across multiple GPUs*

In the next section, we look at multi-worker mirrored strategy.

Multi-worker mirrored strategy

This is a distribution strategy for synchronous training on multiple workers. It inherits from **`tf.distribute.Strategy`**. The following *figure 7.4* shows pre-conditions for a multi-worker strategy:

- multiple GPUs
- multiple workers

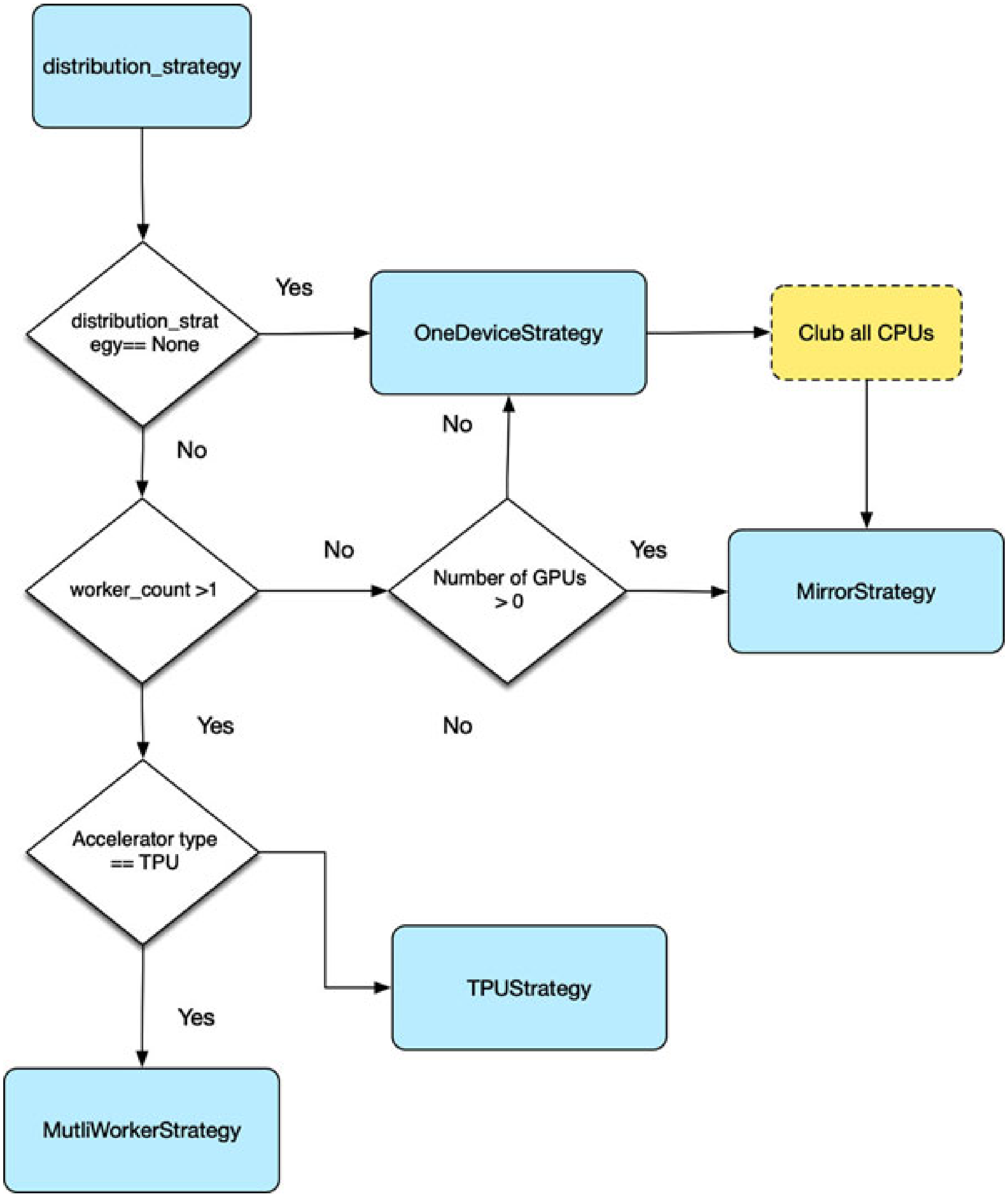


Figure 7.4: *Choosing the multi-worker strategy and TPU strategy*

Next, let us look at various techniques to implement sharing of variables in a multi-worker scenario.

Collective communication in a multi-worker mirrored strategy

It is a parameter implemented in class. `tf.distribute.experimental.CollectiveCommunication`, and used as part of

`MultiWorkerMirroredStrategy` to communicate between GPUs. The following three strategies are available:

- **AUTO**: Automatic runtime choices
- **NCCL**: **`ncclAllReduce`** for reducing the values and ring algorithm for all gather (gathering all values)
- **RING**: **`rng`** all reduce (reducing all values) and ring all gather (collating all values) algorithm

Collective communication routines are common patterns of data transfer among multi-processor setups, especially GPUs. If you have experience with **Message Passing Interface** (**MPI**), then you are already familiar with several collective operations. As an example, the all-reduce routine starts with independent arrays of *N* values on each of *K* processors, and it ends with identical arrays *S* of *N* values, for each processor *k*. Refer to the following *figure 7.5*:

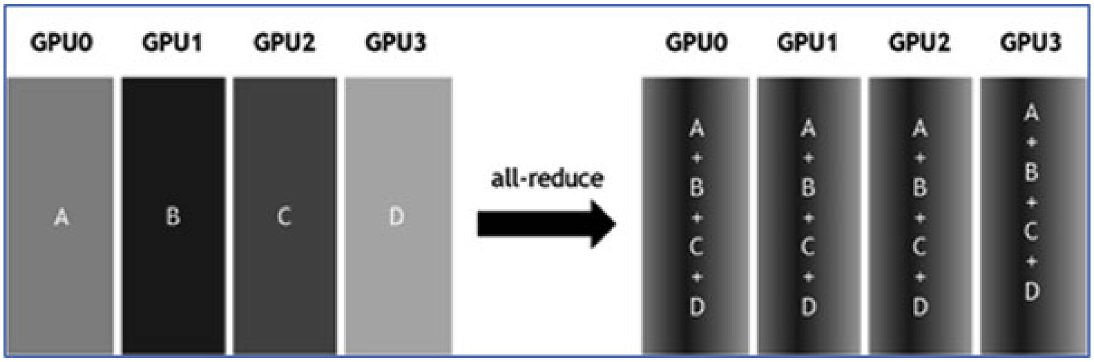


Figure 7.5: *All-reduce routine*

Another common collective routine is **all-gather**. In this routine, each of *K* processors begins with an independent array of *N* values, and collects data from all other processors, and forms a result of dimension *N* K*, as shown in the *figure 7.6*:

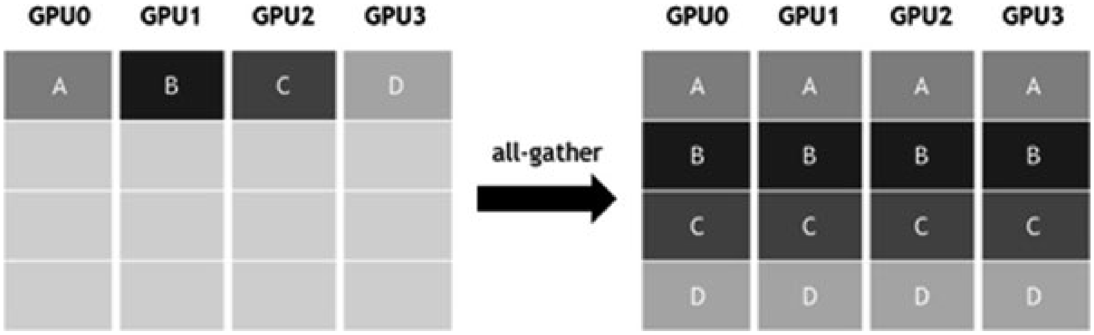


Figure 7.6: *All-gather routine*

In the next section, we will look at **all-reduce algorithm** in more detail.

Ring all reduce algorithm

There are four workers **A**, **B**, **C**, and **D** passing data **a0...a3**, **b0...b3**, **c0...c3**, and **d0...d3**:

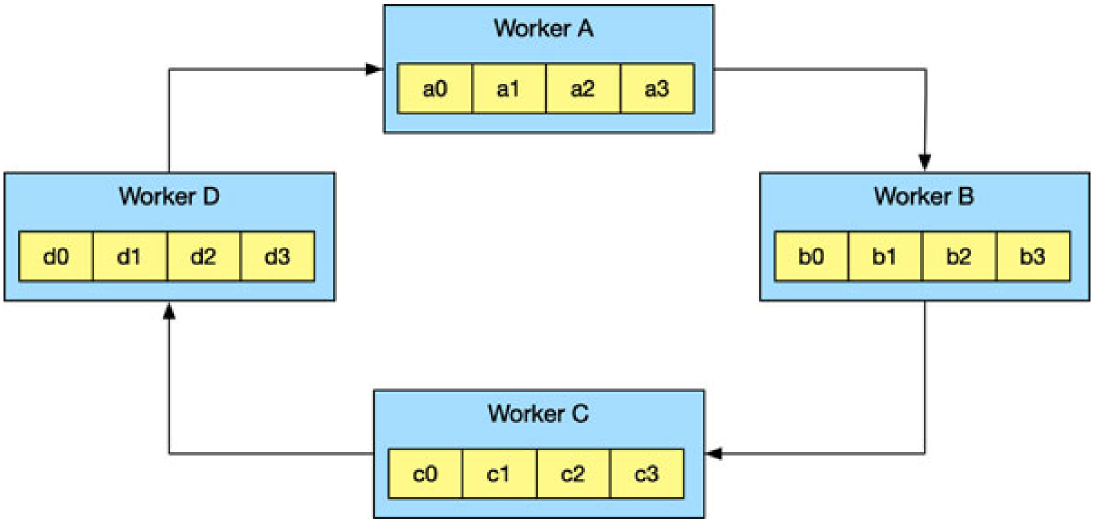


***Figure** 7.7: Communication between various works in a ring all reduce algorithm*

Let us look at each step individually.

The first share-reduce step would have process **A** sending **a0** to process **B**, process **B** sending **b1** to process **C**, and so on. Refer to the following *figure 7.8*:

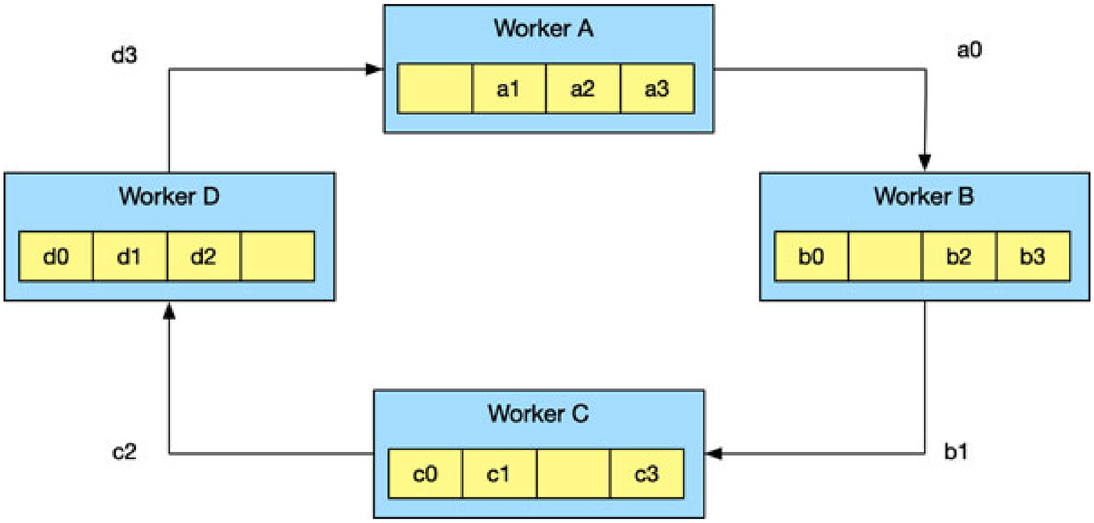


***Figure** 7.8: Communication between various works in a ring of all-reduce algorithm, step 1*

After each process receives the data from the previous process, it applies the **reduce** operator, and then proceeds to send it again to the next process in the ring. Remember that in our sample, the **reduce** operator is a sum, as shown in the *figure 7.9*:

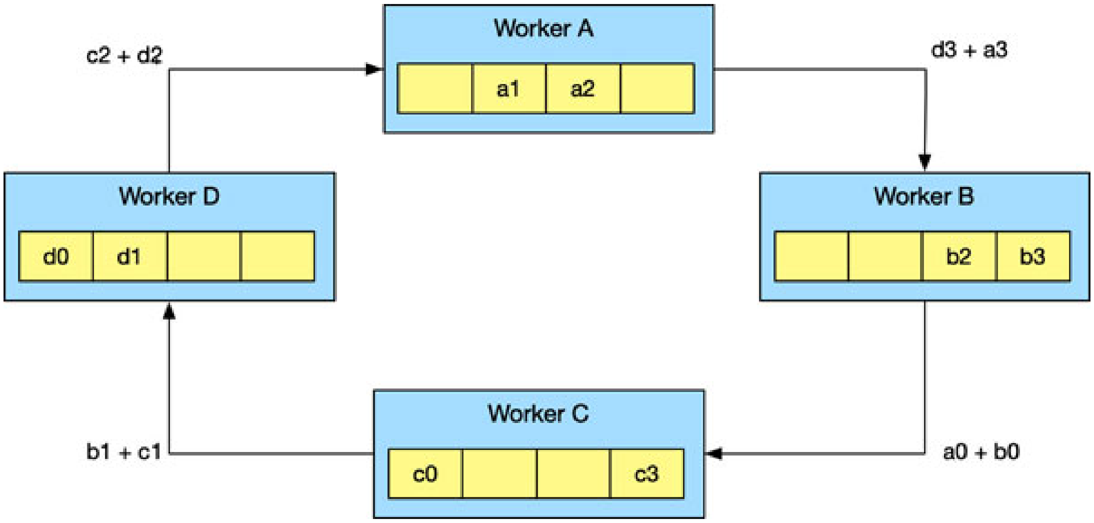


Figure 7.9: *Communication between various works in a ring of all-reduce algorithm, step 2*

Each process sends an intermediate result of the **reduce** operation.

It is important that our reduce operator is *associative* (**https://en.wikipedia.org/wiki/Associative_property**) and *commutative (***https://en.wikipedia.org/wiki/Commutative_property***)* otherwise, you cannot carry along the intermediate result from process to process. The **share-reduce phase** finishes when each process holds the complete reduction of chunk *i*. At this point, each process holds a part of the end result. It visually looks something like this:

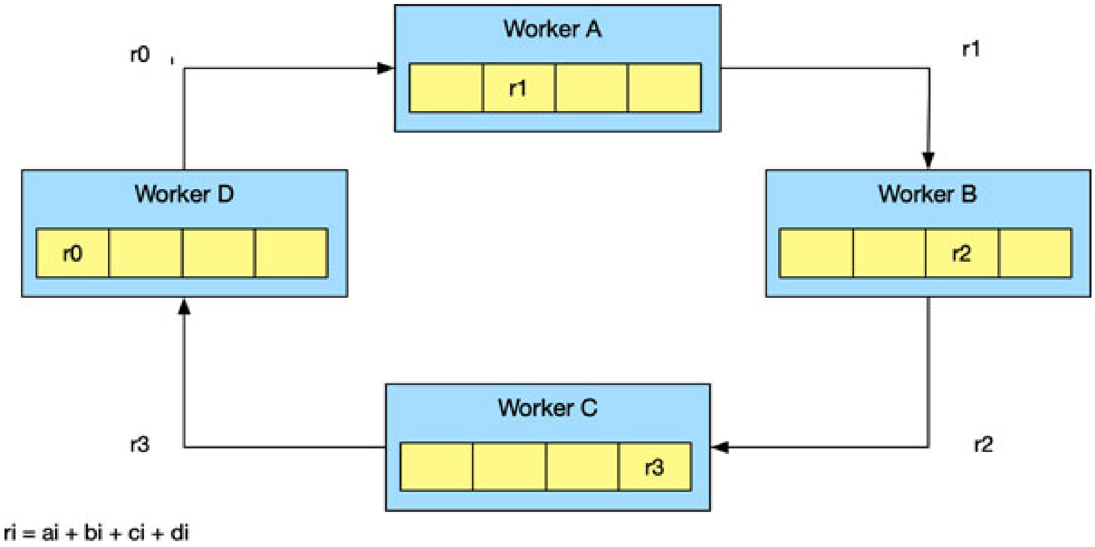


Figure 7.10: *Communication between various works in a ring of all-reduce algorithm, step 3*

The next step is to share the reduced values across four workers as shown in the *figure 7.11*:

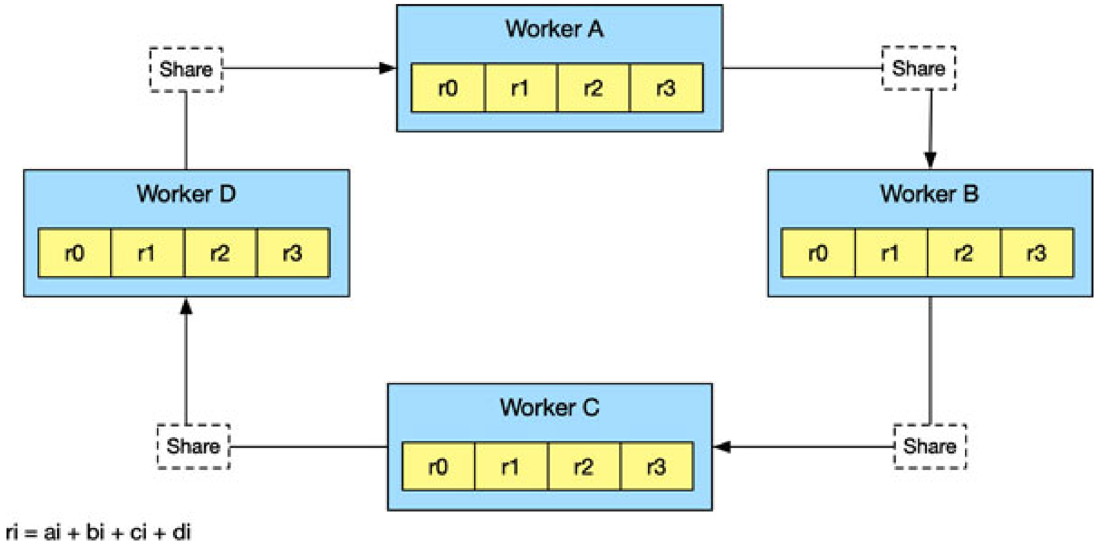


Figure 7.11: *Communication between various works in a ring of all-reduce algorithm, step 4*

Implementation of multi-worker strategy in TensorFlow described previously is implemented using the following class:

```
multiworker_strategy =
tf.distribute.experimental.MultiWorkerMirroredStrategy(
   tf.distribute.experimental.CollectiveCommunication.NCCL)
```

You can also optionally pass the **collective communication strategy**. We passed NCCL in the case.

Code walk-through of MirroredStrategy

Let us do a code walkthrough of the MirroredStrategy implementation

- **MirroredStrategy initialization**: Initialize constructor with a list of GPUs: **`tf.distribute.MirroredStrategy(["GPU:0", "GPU:1"]).`**

  Figure 7.12 shows the sequence of initializing **`MirroredStrategy`**. **`MirroredStrategy`** constructor calls **`MirroredExtended`** which calls **`StrategyExtendedV1`**:

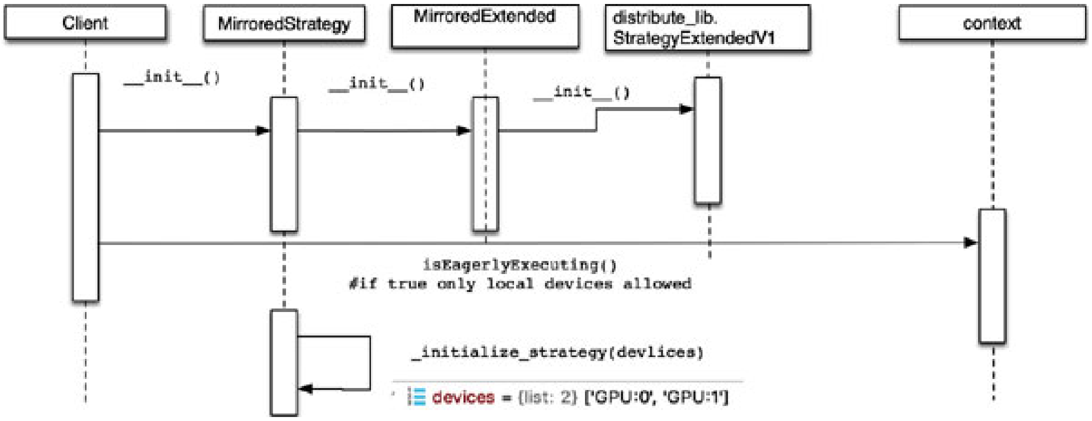


Figure 7.12: *Initialization of mirror strategy*

Both of these are internal classes and not exposed.

Also, a check is made if the graph is eagerly executing. In an **eagerly executing graph**, only local devices are allowed. Figure 7.13 illustrates the initialization for mirror strategy:

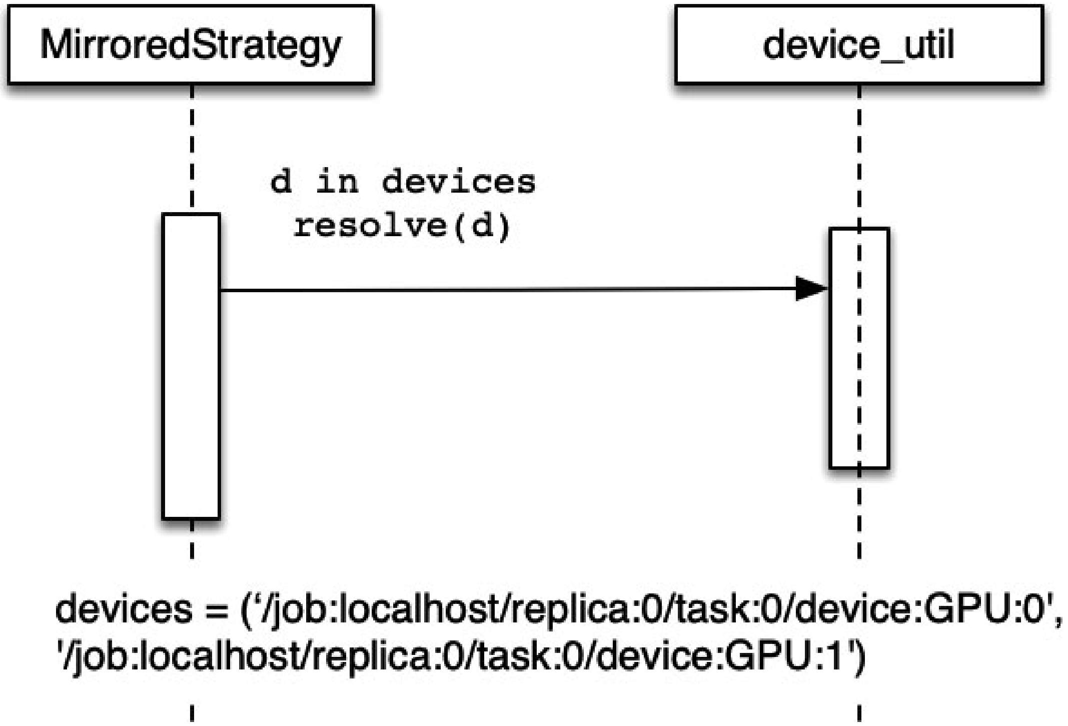

MirroredStrategy
device_util
d in devices
resolve(d)
devices = ('/job:localhost/replica:0/task:0/device:GPU:0',
'/job:localhost/replica:0/task:0/device:GPU:1')

Figure 7.13: *Initialization of mirror strategy: device resolution*

In the next step, we resolve the devices to the full spec as needed by TensorFlow: **hosts**, **replica,** and **tasks**. Figure 7.14 illustrates initialization strategy to be followed depending on the number of workers:

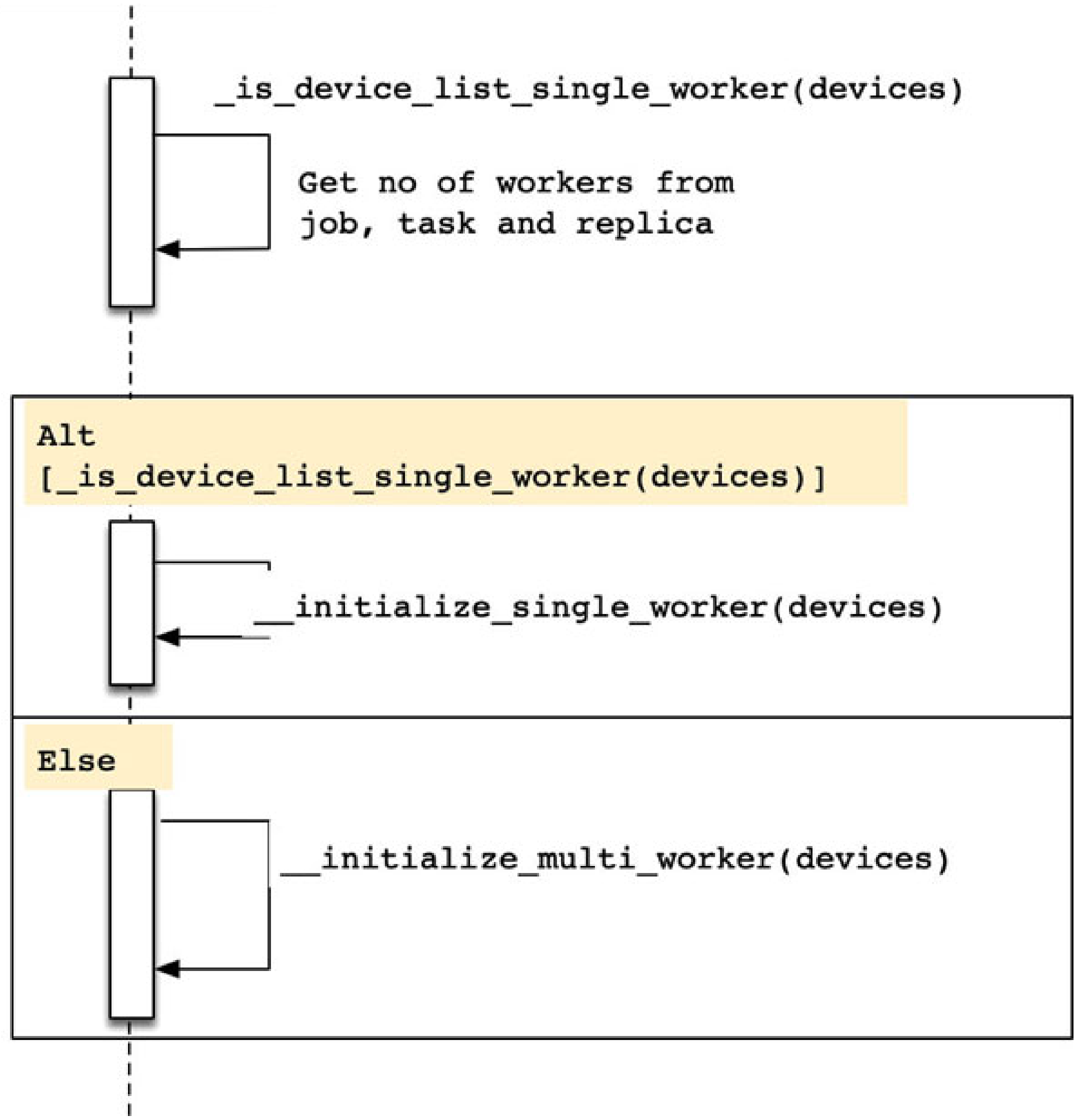

_is_device_list_single_worker(devices)
Get no of workers from job, task and replica
Alt [_is_device_list_single_worker(devices)]
__initialize_single_worker(devices)
Else
__initialize_multi_worker(devices)

Figure 7.14: *Initialization of mirror strategy: device resolution and worker initialization*

A check is made to see if the device list has single or more workers. Next, let us look at the logic behind the initialization of a single worker.

To initialize single worker, a list of devices is passed to it, and it initializes the following variables:

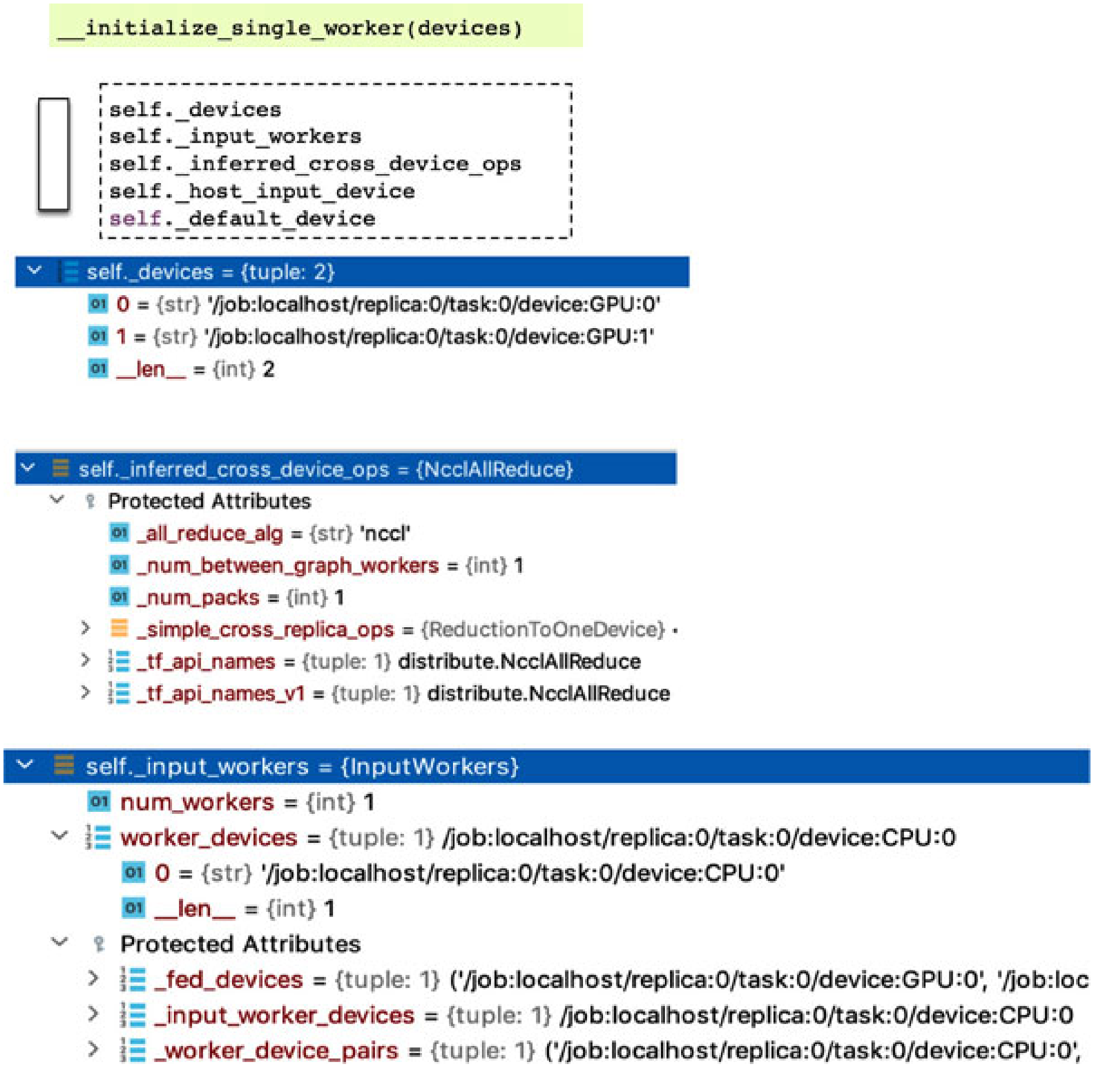
```
self._devices
self._input_workers
self._indferred_cross_device_ops
self._host_input_device
```

In our case, we passed the following devices:

```
('/job:localhost/replica:0/task:0/device:GPU:0',
'/job:localhost/replica:0/task:0/device:GPU:1')
```

`job:localhost` decides that all devices are local. In our case, both the devices are mapped to one worker device `/job:localhost/replica:0/task:0/device:GPU:0`.

Figure 7.15 shows how single worked initialization happens:



```
__initialize_single_worker(devices)

self._devices
self._input_workers
self._inferred_cross_device_ops
self._host_input_device
self._default_device

self._devices = {tuple: 2}
    0 = {str} '/job:localhost/replica:0/task:0/device:GPU:0'
    1 = {str} '/job:localhost/replica:0/task:0/device:GPU:1'
    __len__ = {int} 2

self._inferred_cross_device_ops = {NcclAllReduce}
  Protected Attributes
    _all_reduce_alg = {str} 'nccl'
    _num_between_graph_workers = {int} 1
    _num_packs = {int} 1
  > _simple_cross_replica_ops = {ReductionToOneDevice} ·
  > _tf_api_names = {tuple: 1} distribute.NcclAllReduce
  > _tf_api_names_v1 = {tuple: 1} distribute.NcclAllReduce

self._input_workers = {InputWorkers}
    num_workers = {int} 1
  worker_devices = {tuple: 1} /job:localhost/replica:0/task:0/device:CPU:0
      0 = {str} '/job:localhost/replica:0/task:0/device:CPU:0'
      __len__ = {int} 1
  Protected Attributes
  > _fed_devices = {tuple: 1} ('/job:localhost/replica:0/task:0/device:GPU:0', '/job:loc
  > _input_worker_devices = {tuple: 1} /job:localhost/replica:0/task:0/device:CPU:0
  > _worker_device_pairs = {tuple: 1} ('/job:localhost/replica:0/task:0/device:CPU:0',
```

***Figure 7.15:** Initialization of mirror strategy: single worker initialization*

Inferred cross device ops is inferred as **NcclAllReduce** and is stored in the variable `self._indferred_cross_device_ops`.

- **Set the mirror strategy in the gauge cell**: This value is also set in the gauge cell of the `distribution_strategy_gauge`. Sequence diagram explaining **MirroredStrategy** gauge cell interaction is shown in the *figure 7.16*:

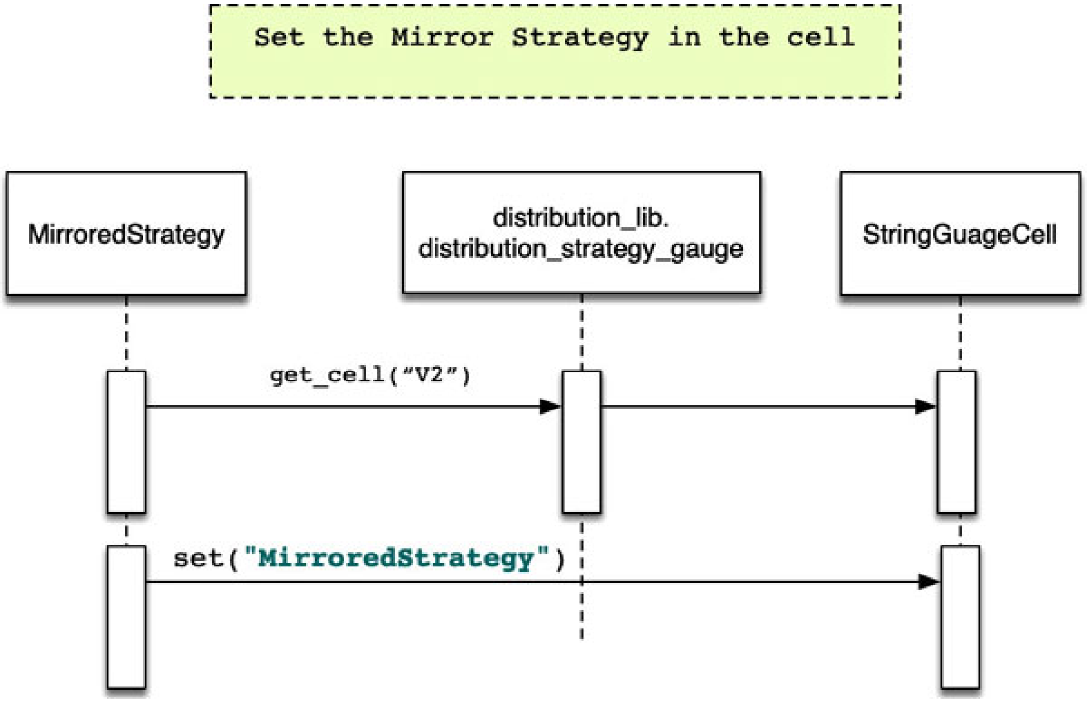


Figure 7.16: *Initialization of mirror strategy: Set mirror strategy in the gauge cell*

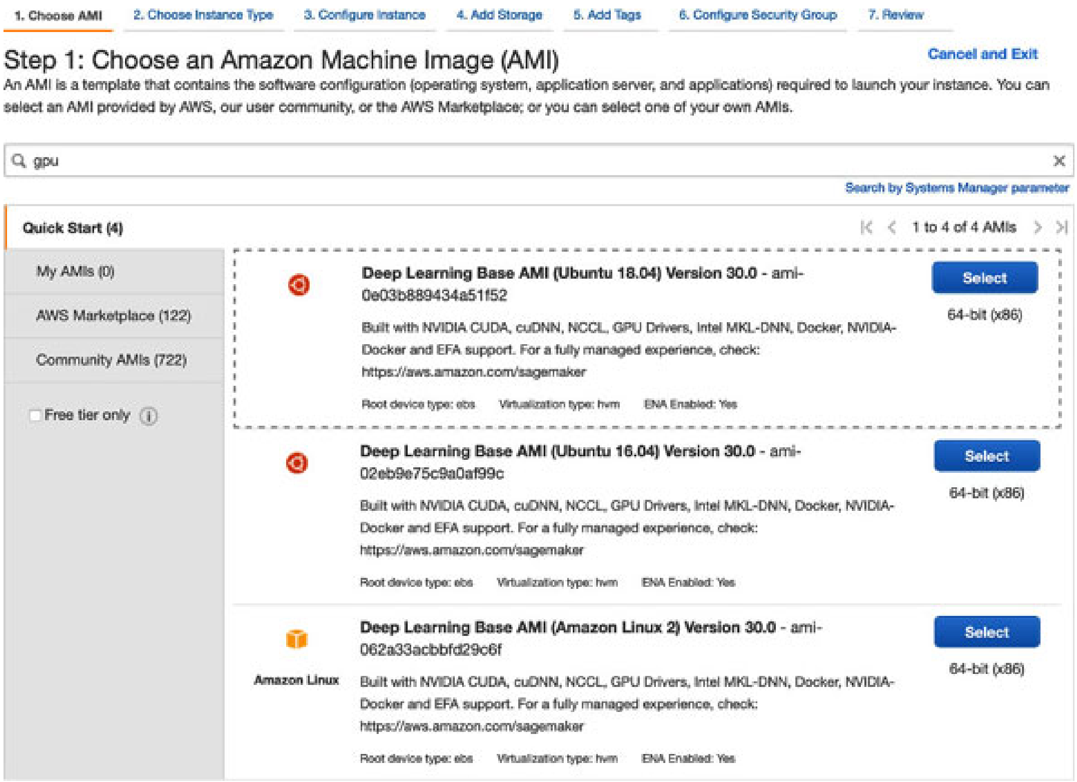
With this, we come to the end of how TensorFlow **MirroredStrategy** gets initialized. Most of these inner workings are opaque to the user of the library.

Running MirroredStrategy on multi CPU AWS virtual machine

Let us create a virtual machine with multiple virtual machines, install TensorFlow and Jupyter, and run MNIST classification with multi-device mirror strategy.

- **Launching a VM**: Follow these steps:

  1. We chose **Ubuntu 18.04 Deep Learning AMI** as shown in the *figure 7.17*:



Step 1: Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

gpu

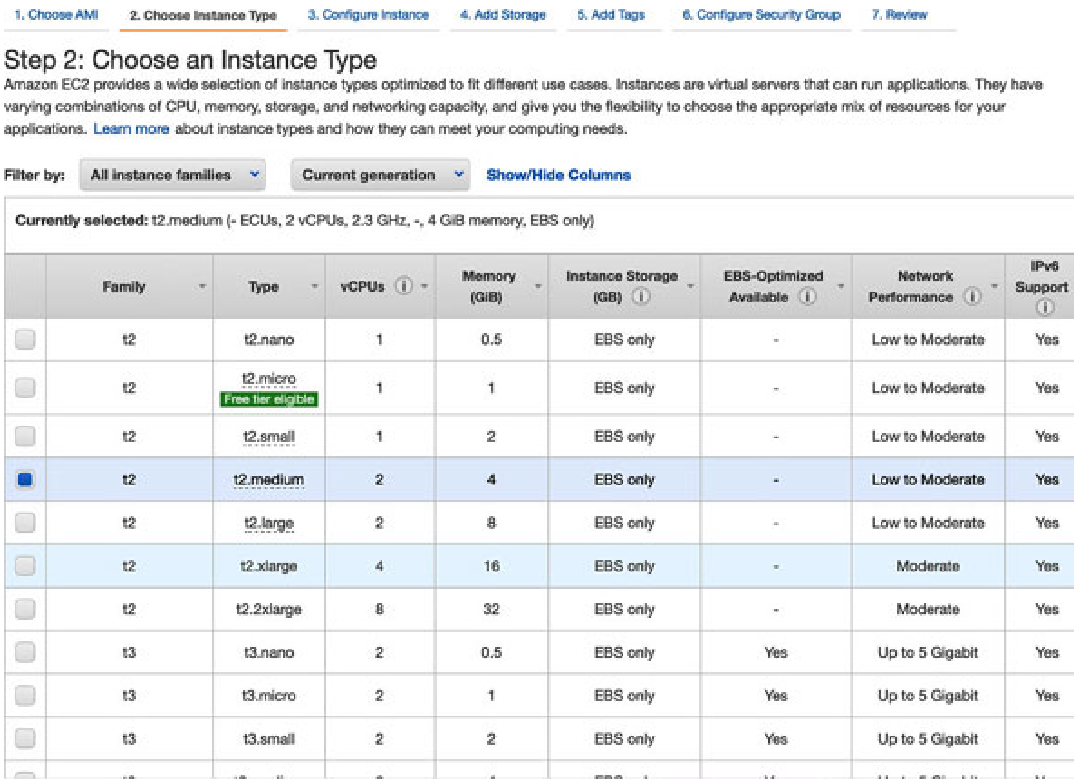
Deep Learning Base AMI (Ubuntu 18.04) Version 30.0 - ami-0e03b889434a51f52
Built with NVIDIA CUDA, cuDNN, NCCL, GPU Drivers, Intel MKL-DNN, Docker, NVIDIA-Docker and EFA support. For a fully managed experience, check: https://aws.amazon.com/sagemaker
Root device type: ebs Virtualization type: hvm ENA Enabled: Yes
Select
64-bit (x86)

Deep Learning Base AMI (Ubuntu 16.04) Version 30.0 - ami-02eb9e75c9a0af99c
Built with NVIDIA CUDA, cuDNN, NCCL, GPU Drivers, Intel MKL-DNN, Docker, NVIDIA-Docker and EFA support. For a fully managed experience, check: https://aws.amazon.com/sagemaker
Root device type: ebs Virtualization type: hvm ENA Enabled: Yes
Select
64-bit (x86)

Amazon Linux
Deep Learning Base AMI (Amazon Linux 2) Version 30.0 - ami-062a33acbbfd29c6f
Built with NVIDIA CUDA, cuDNN, NCCL, GPU Drivers, Intel MKL-DNN, Docker, NVIDIA-Docker and EFA support. For a fully managed experience, check: https://aws.amazon.com/sagemaker
Root device type: ebs Virtualization type: hvm ENA Enabled: Yes
Select
64-bit (x86)

***Figure 7.17:** AMI machine in AWS*

2. After choosing the AMI, choose the instance type as shown in *figure 7.18*:



Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. Learn more about instance types and how they can meet your computing needs.

Filter by: All instance families | Current generation | Show/Hide Columns

Currently selected: t2.medium (- ECUs, 2 vCPUs, 2.3 GHz, -, 4 GiB memory, EBS only)

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
□	t2	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
□	t2	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes
□	t2	t2.small	1	2	EBS only	-	Low to Moderate	Yes
■	t2	t2.medium	2	4	EBS only	-	Low to Moderate	Yes
□	t2	t2.large	2	8	EBS only	-	Low to Moderate	Yes
□	t2	t2.xlarge	4	16	EBS only	-	Moderate	Yes
□	t2	t2.2xlarge	8	32	EBS only	-	Moderate	Yes
□	t3	t3.nano	2	0.5	EBS only	Yes	Up to 5 Gigabit	Yes
□	t3	t3.micro	2	1	EBS only	Yes	Up to 5 Gigabit	Yes
□	t3	t3.small	2	2	EBS only	Yes	Up to 5 Gigabit	Yes

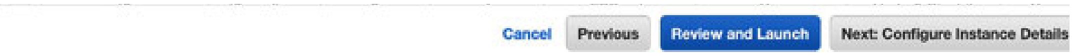
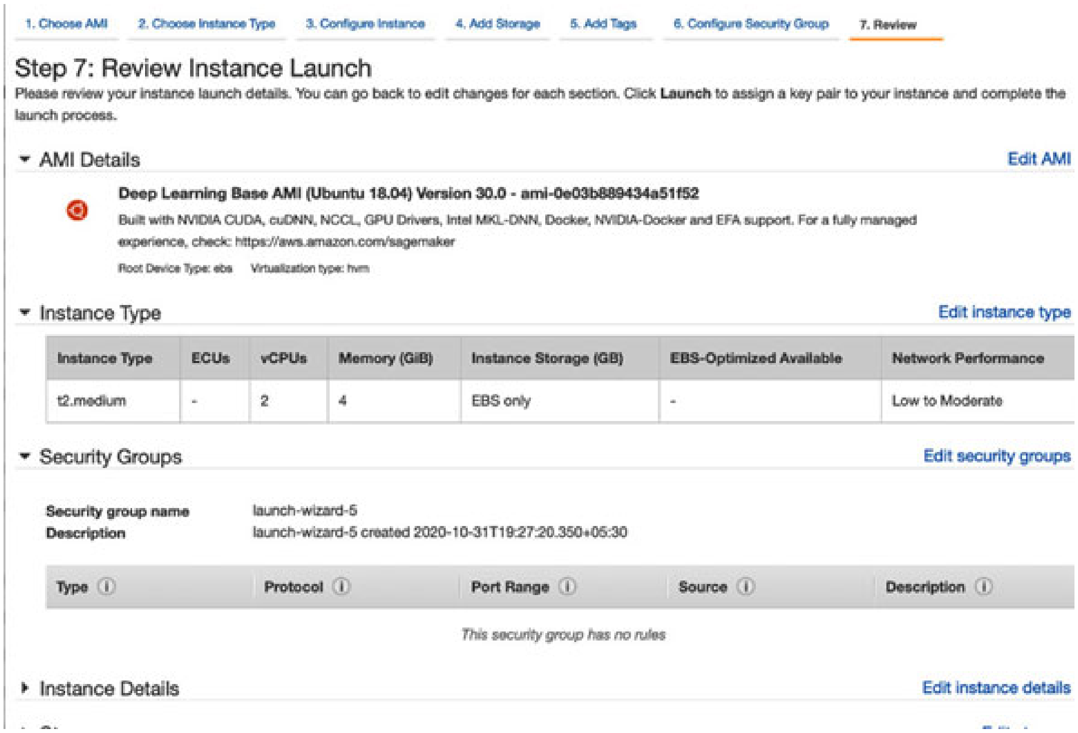


Figure 7.18: *Choose the instance type for the VM, we chose t2.micro*

3. Keep *Steps 3*, *4*, *5*, and *6* default, and go to *Step 7*. Review and launch as shown in *figure 7.19*:



Step 7: Review Instance Launch

Please review your instance launch details. You can go back to edit changes for each section. Click **Launch** to assign a key pair to your instance and complete the launch process.

▾ AMI Details — Edit AMI

Deep Learning Base AMI (Ubuntu 18.04) Version 30.0 - ami-0e03b889434a51f52

Built with NVIDIA CUDA, cuDNN, NCCL, GPU Drivers, Intel MKL-DNN, Docker, NVIDIA-Docker and EFA support. For a fully managed experience, check: https://aws.amazon.com/sagemaker

Root Device Type: ebs   Virtualization type: hvm

▾ Instance Type — Edit instance type

Instance Type	ECUs	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
t2.medium	-	2	4	EBS only	-	Low to Moderate

▾ Security Groups — Edit security groups

Security group name launch-wizard-5
Description launch-wizard-5 created 2020-10-31T19:27:20.350+05:30

Type	Protocol	Port Range	Source	Description

This security group has no rules

▸ Instance Details — Edit instance details

Figure 7.19: *Summary panel before launching the VM on EC2*

4. Once the VM has been launched, SSH into it using the **`.pem`** file:

```
ssh -i rd_app.pem ubuntu@xx.xxx.xxx.xx
```

5. Make sure Conda is downloaded and installed from the following link: **https://docs.anaconda.com/anaconda/install/linux/**

6. Set password for Jupyter. Jupyter will be in **`~/anaconda3/bin`** directory:

```
jupyter config password
```

7. Launch Jupyter as a background process:

```
jupyter notebook&
```

8. Redirect the output of Jupyter and TensorBoard to the local browser:

```
ssh -i rd_app.pem ubuntu@xx.xxx.xxx.xx -L
8889:127.0.0.1:8888
ssh -i rd_app.pem ubuntu@ xx.xxx.xxx.xx -L
6006:127.0.0.1:6006
```

Let us execute the following commands in the Jupyter notebook, in the next section.

- **Define the imports**: Start with defining the imports:

```
import tensorflow_datasets as tfds
import tensorflow as tf
import os
```

First, let us load the dataset using `tensorflow_dataset`.

- **Download the dataset**: We will use `tensorflow_datasets` to load the dataset in `tf.data` format. Loading along with the data is metadata as well. In our case, we had to install first:

```
datasets, info = tfds.load(name='mnist', with_info=True,
as_supervised=True)
mnist_train, mnist_test = datasets['train'], datasets['test']
```

- **Define MirrorStrategy**: Define a `MirrorStrategy` with two devices:

```
strategy = tf.distribute.MirroredStrategy(devices=["/CPU:0",
"/CPU:1"])
```

Output will show the fully qualified names of the devices with the IP address and the replica name and task:

```
  INFO:tensorflow:Using MirroredStrategy with devices
  (‘/job:localhost/replica:0/task:0/device:CPU:0’,
  ‘/job:localhost/replica:0/task:0/device:CPU:1’)
```

Let us print the devices. You can see the output is `2,` in this case for our setup:

```
print('Number of devices:
{}'.format(strategy.num_replicas_in_sync))
    Number of devices: 2
```

Define the number of training and test examples, buffer size, and batch size, per replica:

```
num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples
BUFFER_SIZE = 10000
BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA *
strategy.num_replicas_in_sync
```

Scale function to scale down the image:

```
def scale(image, label):
  image = tf.cast(image, tf.float32)
  image /= 255
  return image, label
```

This will help us define the final training and `test` dataset, which is used for model training:

```
train_dataset =
mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_
SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

- **Define the model with strategy in scope**: The next step is what differentiates distributed training from regular singe device training. Model creation and compilation is done within the **`strategy.scope()`**:

```
with strategy.scope():
 model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(32, 3, activation='relu',
input_shape=(28, 28, 1)),
  tf.keras.layers.MaxPooling2D(),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10)
 ])
 model.compile(loss=tf.keras.losses.SparseCategoricalCrossentrop
 y(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy'])
def decay(epoch):
 if epoch < 3:
 return 1e-3
 elif epoch >= 3 and epoch < 7:
 return 1e-4
 else:
 return 1e-5
class Print_LR(tf.keras.callbacks.Callback):
 def on_epoch_end(self, epoch, logs=None):
 print('\nLearning rate for epoch {} is {}'.format(epoch + 1,
       model.optimizer.lr.numpy()))
checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")
callbacks = [
 tf.keras.callbacks.TensorBoard(log_dir='./logs'),
 tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,
       save_weights_only=True),
 tf.keras.callbacks.LearningRateScheduler(decay),
 Print_LR()
]
```

With all the callbacks defined, let us get down the training of the model.

- **Train the model**: We will call **`model.fit`** to train the model, passing the callbacks defined in the previous section:

  ```
  model.fit(train_dataset, epochs=12, callbacks=callbacks)
  ```

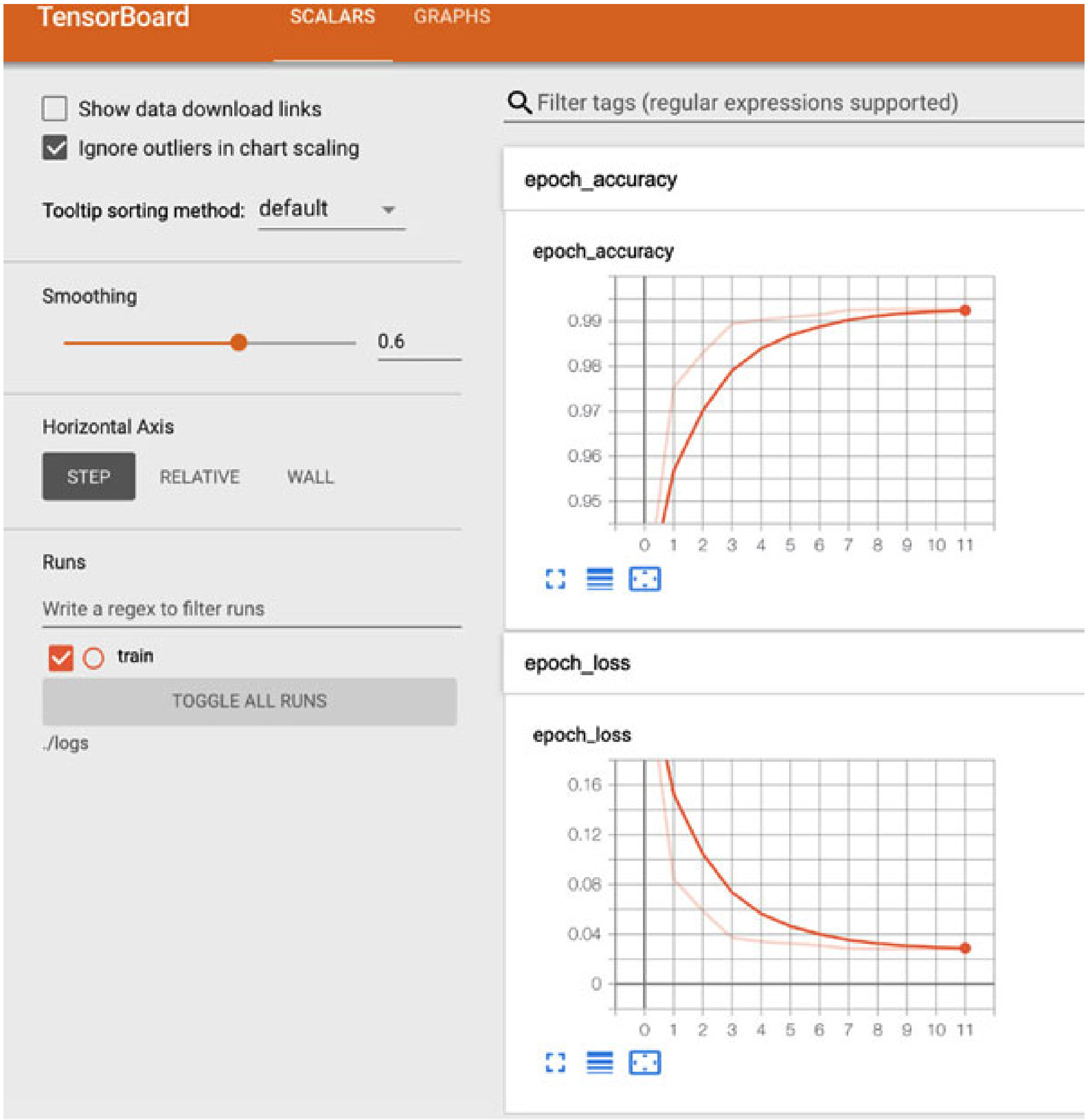
  The output of the various epochs has been shown as follows:

  ```
  Epoch 1/12
  469/Unknown - 24s 51ms/step - accuracy: 0.9259 - loss: 0.2662
  Learning rate for epoch 1 is 0.0010000000474974513
  469/469 [==============================] - 24s 51ms/step -
  accuracy: 0.9259 - loss: 0.2662 - lr: 0.0010
  Epoch 2/12
  469/469 [==============================] - ETA: 0s - accuracy:
  0.9753 - loss: 0.0843
  Learning rate for epoch 2 is 0.0010000000474974513
  469/469 [==============================] - 14s 31ms/step -
  accuracy: 0.9753 - loss: 0.0843 - lr: 0.0010
  469/469 [==============================] - 18s 38ms/step -
  accuracy: 0.9927 - loss: 0.0278 - lr: 1.0000e-05
  Epoch 12/12
  468/469 [============================>.] - ETA: 0s - accuracy:
  0.9929 - loss: 0.0277
  Learning rate for epoch 12 is 9.999999747378752e-06
  469/469 [==============================] - 13s 29ms/step -
  accuracy: 0.9929 - loss: 0.0277 - lr: 1.0000e-05
  ```

  We are able to get an accuracy of **`0.9929`**. Let us look at the training and accuracy as a function of epochs with TensorBoard:

  ```
  !tensorboard --logdir='./logs'
  ```

  The TensorBoard output is shown in the following *figure 7.20*:



□ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: STEP RELATIVE WALL

Runs

Write a regex to filter runs

☑ ○ train

./logs

Figure 7.20: *TensorBoard showing accuracy and loss as a function of epochs*

Multi-worker training with Keras with two processes

- **Overview**: This section demonstrates multi-worker distributed training with Keras model using **`tf.distribute.Strategy`** API, **`tf.distribute.experimental.MultiWorkerMirroredStrategy`**. Using this strategy, a Keras model that has been designed to run on single-worker can seamlessly work on multiple-workers with a very minimal code change. We have modified the original sample available at the following link:

  https://www.tensorflow.org/guide/distributed_training

- **Setup**: First, let us make the necessary imports:

```
import json
import os
import sys
```

Before importing TensorFlow, let us make a few changes to the environment. We will have to disable all GPUs. This will prevent errors caused by the workers, all trying to use the same GPU. For an actual application, each worker would be on a different machine:

```
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

Reset the **TF_CONFIG** environment variable. We will use this later to configure the runtime:

```
os.environ.pop('TF_CONFIG', None)
```

Make sure that the current directory is on Python's path. This will allow the notebook to import the files written by **%%writefile** later:

```
if '.' not in sys.path:
  sys.path.insert(0, '.')
```

Import TensorFlow and check the version:

```
import tensorflow as tf
tf.__version__
'2.7.0'
```

MNIST dataset and model definition

Create an **mnist.py** file with a simple CNN model and dataset loaded using **tf.keras.datasets.mnist.load_data()**. This file will be used by the worker-processes in this example:

```
%%writefile mnist.py
import os
import tensorflow as tf
import numpy as np
def mnist_dataset(batch_size):
  (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
  # The `x` arrays are in uint8 and have values in the range [0, 255].
  # You need to convert them to float32 with values in the range [0,
  1]
  x_train = x_train / np.float32(255)
  y_train = y_train.astype(np.int64)
  train_dataset = tf.data.Dataset.from_tensor_slices(
  (x_train, y_train)).shuffle(60000).repeat().batch(batch_size)
  return train_dataset
def build_and_compile_cnn_model():
  model = tf.keras.Sequential([
    tf.keras.Input(shape=(28, 28)),
    tf.keras.layers.Reshape(target_shape=(28, 28, 1)),
    tf.keras.layers.Conv2D(32, 3, activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
  ])
  model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
    metrics=['accuracy'])
  return model
```

It will create a file **mnist.py**:

```
Writing mnist.py
```

Train the model for a small number of epochs and observe the results of a single worker for verification. With each epoch, the loss drops, and the accuracy increases:

```
import mnist
batch_size = 64
single_worker_dataset = mnist.mnist_dataset(batch_size)
single_worker_model = mnist.build_and_compile_cnn_model()
single_worker_model.fit(single_worker_dataset, epochs=3,
steps_per_epoch=10)
Epoch 1/3
10/10 [==============================] - 0s 39ms/step - loss: 2.2927
- accuracy: 0.1359
Epoch 2/3
10/10 [==============================] - 0s 39ms/step - loss: 2.2841
- accuracy: 0.1781
Epoch 3/3
10/10 [==============================] - 0s 38ms/step - loss: 2.2771
- accuracy: 0.2234
```

Multi-worker configuration with two workers

Let us enter the world of multi-worker training. In TensorFlow, the **TF_CONFIG** environment variable is required for training on multiple machines, each of which possibly has a different role. **TF_CONFIG** is a JSON string used to specify the cluster configuration on each worker that is part of the cluster. Example configuration:

```
tf_config = {
  'cluster': {
   'worker': ['localhost:11111', 'localhost:22222']
  },
  'task': {'type': 'worker', 'index': 0}
}
```

`TF_CONFIG` serialized as a JSON string:

```
json.dumps(tf_config)
'{"cluster": {"worker": ["localhost:11111", "localhost:22222"]},
"task": {"type": "worker", "index": 0}}'
```

There are two components of **`TF_CONFIG`**: **cluster** and **task**

- **cluster** is the same for all workers and provides information about the training cluster, which is a dict consisting of different types of jobs, such as worker. In multi-worker training with **`MultiWorkerMirroredStrategy`**, there is usually one worker that takes on a little more responsibility like saving checkpoint and writing summary file for TensorBoard in addition to what a regular worker does. Such a worker is referred to as the chief worker, and it is customary that the worker with index **0** is appointed as the chief worker (in fact, this is how **`tf.distribute.Strategy`** is implemented).
- **task** provides information of the current task and is different on each worker. It specifies the type and index of that worker.

In this example, we will set the task type to **`worker`** and the task index to `0`. This machine is the first worker. It will be appointed as the chief worker and does more work than the others.

Note that other machines will need to have the **`TF_CONFIG`** environment variable set as well, and it should have the same cluster dict, but different task type or task index, depending on what the roles of those machines are.

For illustration purposes, this sample shows how one may set a **`TF_CONFIG`** with two workers on localhost. In practice, users would create multiple workers on external IP addresses/ports and set **`TF_CONFIG`** on each worker appropriately.

In this example, you will use two workers, the first worker's **`TF_CONFIG`** is shown earlier. For the second worker, you would set **`tf_config['task']['index']=1`**.

The preceding **`tf_config`** is just a local variable in Python. To use it to configure training, this dictionary needs to be serialized as JSON, and placed in the **`TF_CONFIG`** environment variable.

Environment variables and subprocesses in notebooks

Subprocesses inherit environment variables from their parents. So, if you set an environment variable in this Jupyter notebook process:

```
os.environ['GREETINGS'] = 'Hello TensorFlow!'
```

You can access the environment variable from a subprocess:

```
%%bash
echo ${GREETINGS}
Hello TensorFlow!
```

In the next section, you'll use this to pass the **`TF_CONFIG`** to the worker subprocesses. You would never really launch your jobs this way, but it's sufficient for the purposes of this tutorial, to demonstrate a minimal multi-worker example.

Strategy to be chosen

In TensorFlow, there are two main forms of distributed training as we saw earlier in the chapter:

- **Synchronous training**, where the steps of training are synced across the workers and replicas.
- **Asynchronous training**, where the training steps are not strictly synced.

`MultiWorkerMirroredStrategy`, which is the recommended strategy for synchronous multi-worker training, will be demonstrated in this guide. To train the model, use an instance of **`tf.distribute.experimental.MultiWorkerMirroredStrategy`**.

`MultiWorkerMirroredStrategy` creates copies of all variables in the model's layers on each device across all workers. It uses **`CollectiveOps`**, a TensorFlow operation for collective communication, to aggregate gradients, and keep the variables in sync.

`TF_CONFIG` is parsed and TensorFlow's GRPC servers are started at the time `MultiWorkerMirroredStrategy()` is called, so the `TF_CONFIG` environment variable must be set before a `tf.distribute.Strategy` instance is created. Since `TF_CONFIG` is not set yet, this strategy is effectively single-worker training.

Train the model

With the integration of **`tf.distribute.Strategy`** API into **`tf.keras`**, the only change you will make to distribute the training to multiple-workers is enclosing the model building and **`model.compile()`** call inside **`strategy.scope()`**. The distribution strategy's scope dictates how and where the variables are created. In the case of **`MultiWorkerMirroredStrategy`**, the variables created are **`MirroredVariables`**, and they are replicated on each of the workers.

> **Currently, there is a limitation in `MultiWorkerMirroredStrategy` where TensorFlow ops need to be created after the instance of strategy is created. If you see `RuntimeError:` collective ops must be configured at program startup, try creating the instance of `MultiWorkerMirroredStrategy` at the beginning of the program and put the code that may create ops after the strategy is instantiated.**

To run with **`MultiWorkerMirroredStrategy`**, you'll need to run worker processes and pass a **`TF_CONFIG`** to them.

Like the **`mnist.py`** file written earlier, here is the **`main.py`** that each of the workers will run:

```
%%writefile main.py
import os
import json
import tensorflow as tf
import mnist
per_worker_batch_size = 64
tf_config = json.loads(os.environ['TF_CONFIG'])
num_workers = len(tf_config['cluster']['worker'])
strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
global_batch_size = per_worker_batch_size * num_workers
multi_worker_dataset = mnist.mnist_dataset(global_batch_size)
with strategy.scope():
  # Model building/compiling need to be within `strategy.scope()`.
  multi_worker_model = mnist.build_and_compile_cnn_model()
multi_worker_model.fit(multi_worker_dataset, epochs=3,
steps_per_epoch=70)
```

In the preceding code snippet, note that the **`global_batch_size`**, which gets passed to **`dataset.batch`**, is set to **`per_worker_batch_size * num_workers`**. This ensures that each worker processes batches of **`per_worker_batch_size`** examples regardless of the number of workers.

The current directory now contains both Python files:

```
%%bash
ls *.py
```

You will see the list of Python files generated:

```
main.py
mnist.py
```

JSON serializes the **`TF_CONFIG`** and adds it to the environment variables:

```
os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

Now, launch a worker process that will run the **`main.py`** and use the **`TF_CONFIG`**:

```
# first kill any previous runs
%killbgscripts
```

All background processes were killed:

```
%%bash --bg
python main.py &> job_0.log
Starting job # 0 in a separate thread.
```

There are a few things to note about the preceding command:

- It uses the **`%%bash`** which is a notebook magic (**https://ipython.readthedocs.io/en/stable/interactive/magics.html**) to run the bash commands.
- It uses the **`--bg`** flag to run the bash process in the background, because this worker will not terminate. It waits for all the workers before it starts.

The backgrounded worker process won't print output to this notebook. So, the **`&>`** redirects its output to a file, and you can see what happened.

We will wait a few seconds for the process to start up:

```
import time
time.sleep(5)
```

Worker's logfile so far:

```
%%bash
cat job_0.log
2020-11-07 03:07:49.970694: I
tensorflow/stream_executor/platform/default/dso_loader.cc:48]
Successfully opened dynamic library libcudart.so.10.1
..
2020-11-07 03:07:51.690832: I
tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:301]
Initialize GrpcChannelCache for job worker -> {0 -> localhost:12345,
1 -> localhost:23456}
2020-11-07 03:07:51.691186: I
tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:405]
Started server with target: grpc://localhost:12345
```

The last line of the log file should say: **`Started server with target: grpc://localhost:12345`**.

- The first worker is now ready and is waiting for all the other worker(s) to be ready to proceed.

  Update the **`tf_config`** for the second worker›s process to pick up:

```
 tf_config['task']['index'] = 1
 os.environ['TF_CONFIG'] = json.dumps(tf_config)
```

- Now, launch the second worker. This will start the training since all the workers are active (so there's no need to background this process):

```
%%bash
python main.py
Epoch 1/3
70/70 [==============================] - 10s 142ms/step - loss:
2.2582 - accuracy: 0.1652
Epoch 2/3
70/70 [==============================] - 11s 153ms/step - loss:
2.1708 - accuracy: 0.2890
Epoch 3/3
70/70 [==============================] - 10s 148ms/step - loss:
2.0728 - accuracy: 0.4189
..
2020-11-07 03:08:22.942543: I
tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:301]
Initialize GrpcChannelCache for job worker -> {0 ->
localhost:12345, 1 -> localhost:23456}
2020-11-07 03:08:22.942831: I
tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:405]
Started server with target: grpc://localhost:23456
input: "Placeholder/_0"
input: "Placeholder/_1"
attr {
 key: "Toutput_types"
 value {
 list {
  type: DT_FLOAT
  type: DT_INT64
 }
 }
}
attr {
 key: "output_shapes"
 value {
 list {
  shape {
  dim {
  size: 28
  }
  dim {
  size: 28
  }
  }
  shape {
  }
 }
 }
}
```

Now, if you recheck the logs written by the first worker, you'll see that it participated in training that model:

```
%%bash
cat job_0.log
2020-11-07 03:07:49.970694: I
tensorflow/stream_executor/platform/default/dso_loader.cc:48]
Successfully opened dynamic library libcudart.so.10.1
…
2020-11-07 03:07:51.690832: I
tensorflow/core/distributed_runtime/rpc/grpc_channel.cc:301]
Initialize GrpcChannelCache for job worker -> {0 ->
localhost:12345, 1 -> localhost:23456}
2020-11-07 03:07:51.691186: I
tensorflow/core/distributed_runtime/rpc/grpc_server_lib.cc:405]
Started server with target: grpc://localhost:12345
input: "Placeholder/_0"
input: "Placeholder/_1"
attr {
 key: "Toutput_types"
 value {
 list {
  type: DT_FLOAT
  type: DT_INT64
 }
 }
}
attr {
 key: "output_shapes"
 value {
 list {
  shape {
```

```
  dim {
   size: 28
  }
  dim {
   size: 28
  }
  }
  shape {
  }
 }
 }
}
Epoch 1/3
70/70 [==============================] - 10s 142ms/step - loss:
2.2582 - accuracy: 0.1652
Epoch 2/3
70/70 [==============================] - 11s 153ms/step - loss:
2.1708 - accuracy: 0.2890
Epoch 3/3
70/70 [==============================] - 10s 148ms/step - loss:
2.0728 - accuracy: 0.4189
```

Unsurprisingly, this ran slower than the test run at the beginning of this tutorial. Running multiple workers on a single machine only adds overhead. The goal here was not to improve the training time, but only to give an example of multi-worker training.

```
# Delete the `TF_CONFIG` and kill any background tasks so they don't
affect the next section.
os.environ.pop('TF_CONFIG', None)
%killbgscripts
All background processes stopped
```

As can be seen here, multiple processes as part of **`MultipleWorkerStrategy`** coordinate the training through GRPC communication.

Conclusion

In this chapter, we did a deep dive on distributed training with TensorFlow. We started with understanding the classes like **`MirrorStrategy`** and **`MultiWorkerMirrorStrategy`**. We looked at the inner workings of **`MirrorStrategy`**, for example, decision on whether it will be single or **`MultiWorkerMirrorStrategy`**. We also looked at multi-CPU AWS VM with two

CPU devices for a `MirrorStrategy`. In the end, we looked at distributed training coordinated using `GRPC` and **TF**`_CONFIG` playing a vital role in helping workers discover each other.

Questions

1. What is the difference between MirrorStrategy and MultiWorkerMirrorStrategy?
2. How many GPUs can a machine have?
3. Where do you specify the host and port of workers?
4. What are the three strategies available in MultiWorkerMirroredStrategy to communicate between GPUs?

References

- **TensorFlow Core**: https://www.tensorflow.org/guide/.
- **Fast Multi-GPU collectives with NCCL**: https://developer.nvidia.com/blog/fast-multi-gpu-collectives-nccl/.
- **NVIDIA Collective Communication Library (NCCL) Documentation**: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html.

CHAPTER 8

Reinforcement Learning

Introduction

Reinforcement learning is a field of machine learning that deals with the problem of how an intelligent agent can learn how to make a good sequence of decisions. This is different from the other agents as we are talking about not one decision but a sequence of decisions.

An example of reinforcement is when an agent is learning to play a breakout game from *Atari,* using the pixels as inputs. *Figure 8.1* helps understand the interactions of action, observation, and reward between an agent and the environment in a reinforcement learning setup:

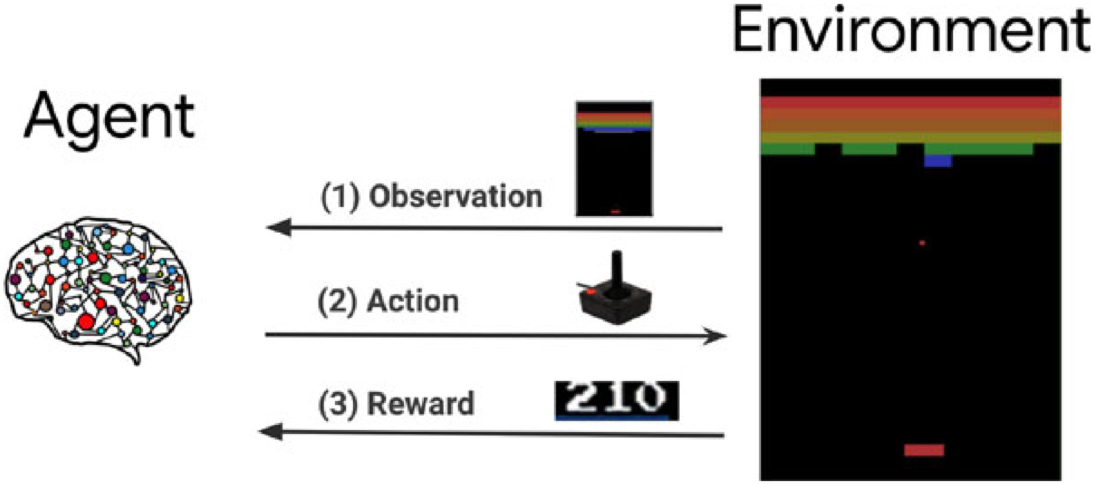


Figure 8.1: *Interactions in a reinforced learning*

Image sourced from *tensorflow.org* under creative commons license:

https://www.tensorflow.org/agents/tutorials/0_intro_rl

The four key concepts of reinforcement learning are as follows:

- **Optimization**: The goal is to find an optimal way to make decisions.
- **Delayed consequences**: The decisions taken now can affect much later. Decisions have long term ramifications in addition to short term benefits. This leads to delayed credit assignment.

- **Exploration**: An agent explores its environment and tries to learn from it, where it learns only if it tries.
- **Generalization**: Policy is mapping from the past experiences to actions. Generalization provides a higher level understanding of the task and helps learn from the past behavior.

Let us compare reinforcement learning to other types in machine learning:

- **AI planning**: Automated planning and scheduling, also denoted as simply AI planning, is a branch of artificial intelligence that is concerned with the realization of strategies or action sequences, usually for execution by intelligent agents, autonomous robots, and unmanned vehicles. Planning is also related to decision theory. *Table 8.1* shows properties of AI planning, that is, which of the areas listed in rows does it support:

	AI planning	**SL**	**UL**	**RL**
Optimization	X			
Learns from experience				
Generalization	X			
Delayed consequences	X			
Exploration				

Table 8.1: *Properties of AI planning*

- **Supervised learning and unsupervised learning versus reinforcement learning**: In supervised learning, we do optimization and generalization but the labels are predefined unlike reinforcement learning. RL is provided the censored labels. *Table 8.2* illustrates comparison between AI planning, supervised learning, unsupervised learning, and reinforcement learning:

	AI planning	**SL**	**UL**	**RL**
Optimization	X			x
Learns from experience		X	x	x
Generalization	x	X	x	x
Delayed consequences	x			x
Exploration				x

Table 8.2: *Properties of various machine learning techniques compared*

  - **SL** – Supervised learning
  - **UL** – Unsupervised learning

  - **RL** – Reinforcement learning
  - **IL** – Imitation learning

  Imitation learning involves optimization, generalization, and delayed consequences. Learn from the experiences of others.

- **Unsupervised learning versus reinforcement learning**: In unsupervised learning, the model involves optimization and generalization, but does not involve exploration or delayed consequences. There are no labels from the real world.

Structure

In this chapter, we will cover the following topics:

- Sequential decision process
- Markov decision process
- Model free policy
- Monte Carlo method
- DQN networks
- Atari game with DQN network
- Introducing TF-agents
- CartPole game with DQN networking using TF-agents
- SAC agent support in TF-agents

Objective

The objective of this chapter is to introduce the basics of reinforcement learning, and compare it with other deep learning approaches. This will be followed by concepts like policy evaluation, policy iteration, Markov decision process, and model free learning. We will also look at the sample implementations in Python. Then, we will move to *Q learning* with the DQN networks and SAC agent algorithm. Towards the latter half of the chapter, we will look at the new library on reinforcement learning called **TF-agents**.

Sequential decision process: agent and the world

In sequential decision processing, at each time step, the agent makes a decision a_t based on a decision and gets a reward r_t and observation o_t from the environment.

Figure 8.2 shows flow of observation, action, and reward between **Agent** and **Environment**:

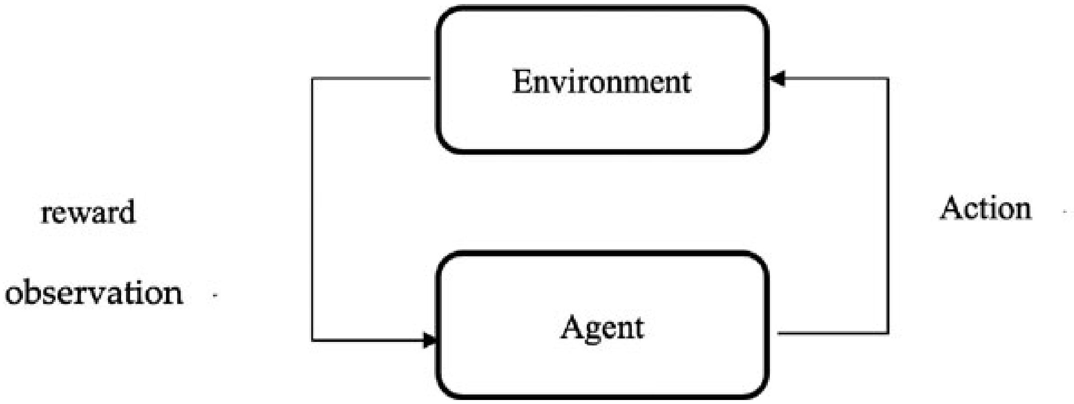


Figure 8.2: *Environment and agent interaction in reinforcement learning*

History is a past sequence of actions, rewards, and observations used by the agent to make decisions into the future:

$$h=(a_1,r_1,o_{1,}a_2,r_2,o_{2,}a_2,r_2,o_2)$$

State is a function of history $s_t=(h_t)$. There is also a world state which might be hidden from the agent. Assuming that all the RL algorithms make is the current state S_t, has sufficient information about history, and it is alone sufficient to make prediction into the future state S_{t+1}. We call the state S_t Markov:

$$p(s_{t+1}|s_t,a_t)=p(s_{t+1}|h_t,a_t)$$

Future is dependent on the present and not the past. To reduce complexity, and instead of state being history, it is assumed that state is equal to observation at that time:

$$s_t=o_t$$

Full observable Markov Decision Process (MDP): If the preceding statement is true, that is, observability is the environment and agent state, then the process is referred to as **Markov Decision Process**.

Partially Observable Markov Decision Process (POMDP): A **partially observable Markov decision process** (**POMDP**) is a generalization of a MDP (**https://en.wikipedia.org/wiki/Markov_decision_process**). A POMDP models an agent-based decision process where it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the state of the environment. Instead, it must maintain a model (the probability distribution of different observations given the underlying state) and the underlying MDP. Unlike the policy function in MDP, which maps the states to the actions, POMDP's policy is a mapping from the observations (or belief states) to the actions.

If the agent does not consider $s_t = o_t$, then we use history h_t or other sources like RNN to create state s_t.

Types of Markov Decision Process

First decision process is referred to as bandits, where it is assumed that actions have no influence over the observations.

In MDP and POMDP described earlier, the actions affect future observations. World changes in two ways depending on the history and actions:

- **Deterministic**: Given the history and action, there is a single reward and observation.
- **Stochastic**: Given the history and action, there are many different rewards and observations possible.

Model in reinforcement learning

Model predicts the next state and rewards from that potential state. Transition model predicts the next state s' given $s_t = s, a_1 = a$:

$$p\left(s_{t+1} = s' \mid s_t = s, a_t = a\right)$$

Reward model predicts reward r_t given $s_t = s, a_1 = a$:

$$r\left(s_t = s, a_t = a\right) = \mathbb{E}\left[r_t \mid s_t = s, a_t = a\right]$$

Policy in reinforcement learning

Policy π determines the action selected given a state s_t. There are two types of policies as follows:

- **Deterministic policy**: It means there is one action per state:

$$\pi(s) = a$$

- **Stochastic policy**: It means distribution of actions given a state:

$$\pi(a|s) = \Pr\left(a_t = a | s_t = s\right)$$

Next, we look at the *value* function.

Value function

Value function V^{π} is the expected sum of discounted future rewards for a given policy π:

$$V^{\pi}\left(s_t = s\right) = \mathbb{E}_{\pi}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \cdots | s_t = s\right]$$

The preceding equation shows the value function equation. Discounted factor γ determines the immediate versus future rewards $\gamma \in (0, 1)$.

Types of RL agents

RL agents can be policy based, value function based, or model based**.** They have a model or an explicit policy or a value function. Some RL agents could be a combination of value function or policy based or as shown in *figure 8.3*:

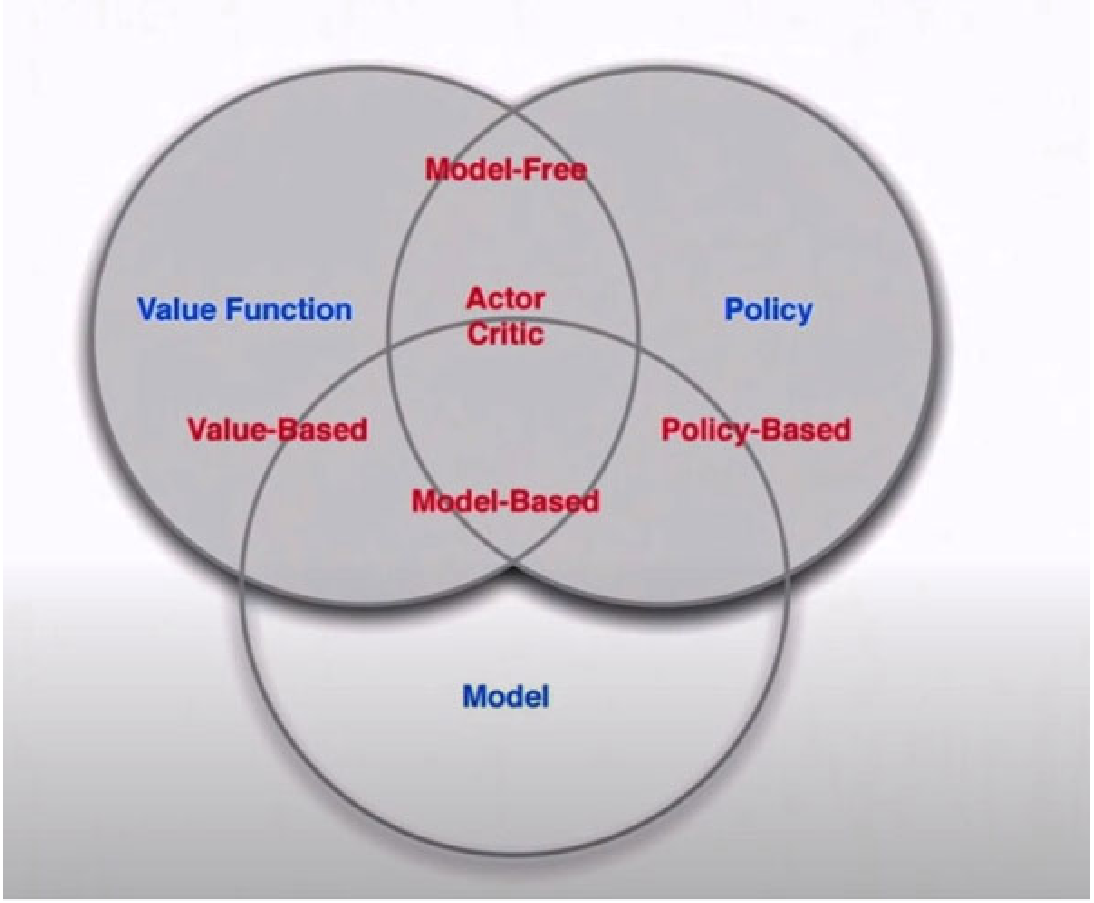


Figure 8.3: *Different types of reinforcement learning techniques*

Source: *David Sliver's RL course: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver*

A lot of algorithms are at intersection of these types, for example, *Actor Critic* is a combination of value and policy-based model.

Markov decision process

Markov decision process can be defined as a finite set of states S, where *P* is the transition model that specifies .

Finite number of states can be presented as the following matrix:

$$P = \begin{pmatrix} P(s_1|s_1) & P(s_2|s_1) & \cdots & P(s_N|s_1) \\ P(s_1|s_2) & P(s_2|s_2) & \cdots & P(s_N|s_2) \\ \vdots & \vdots & \ddots & \vdots \\ P(s_1|s_N) & P(s_2|s_N) & \cdots & P(s_N|s_N) \end{pmatrix}$$

Model free policy

Initialize policy π, calculate the $Q^{\pi} = Q(s, a)$, update the π value and iterate.

Monte Carlo on policy Q evaluation

- Initialize $N(s,a) = 0, G(s,a) = 0, Q^{\pi}(s,a) = 0, \forall s \in S, \forall a \in A$:
    - *N* is a count of state actions.
    - *G* is the sum of all rewards in the past across all episodes.
    - *Q is the q* value.
- Loop for evaluating:
    - Using policy sample episode

      $i = s_{i,1}, a_{i,1}, r_{i,1}, s_{i,2}, a_{i,2}, r_{i,2}, \ldots, s_{i,T_i}, G_{i,t} = r_{i,t} + \gamma r_{i,t+1} + \gamma^2 r_{i,t+2} + \cdots \gamma^{T_i - 1} r_{i,}$
    - Suppose, both the environment transitions and policy are stochastic. In this case, the probability of a step trajectory is for first or every time that is visited in episode.
- $N(s,a) = N(s,a) + 1$, $G(s,a) = G(s,a) + G_{(i,t)}$

  Update estimate $Q^{\pi}(s,a) = G(s,a) / N(s,a)$

  Given an estimate $Q(^{\pi}i)(s,a)$ $\forall$ s,a

  Update new policy:

$$\pi_{(i+1)}(s) = arg\ (max)\ Q^{(\pi i)}(s,a)$$

Till now, we have dealt only with deterministic policies, where states map to actions. We need to add stochasticity to the policy else the policy will map to only one action.

The idea is to balance exploration and exploitation by being random.

Let be the number of actions. Then, an -greedy policy with respect to a *state-action value Q^π (s, a)* is:

$$\pi(a \mid s) = \begin{cases} 1-\epsilon, \max_a Q^\pi(s,a) \\ a, \epsilon / |A| \end{cases}$$

Pseudo code for the same is:

```
random ()
if :
  pull random action else:
else:
  pull current-best action
```

Let us implement this algorithm in Python to get a better understanding:

1. First, we implement the **`Actions`** class. It has three class level variables – **`m`**, **`mean`**, and **`N`** (number of iterations):

```
class Actions:
   def __init__(self, m):
    self.m = m
    self.mean = 0
    self.N = 0
   # Choose a random action
   def choose(self):
    return np.random.randn() + self.m
   # Update the action-value estimate
   def update(self, x):
    self.N += 1
    self.mean = (1 - 1.0 / self.N)*self.mean + 1.0 / self.N * x
```

2. Let us define a method which takes three values for **`m`**, a value for epsilon, and number of iterations **`N`**:

```
def run_experiment(m1, m2, m3, eps, N):
   actions = [Actions(m1), Actions(m2), Actions(m3)]
   data = np.empty(N)
   for i in range(N):
    # epsilon greedy
    p = np.random.random()
    if p < eps:
       j = np.random.choice(3)
    else:
       j = np.argmax([a.mean for a in actions])
    x = actions[j].choose()
    actions[j].update(x)
    # for the plot
    data[i] = x
   cumulative_average = np.cumsum(data) / (np.arange(N) + 1)
# plot moving average ctr
plt.plot(cumulative_average)
plt.plot(np.ones(N)*m1)
plt.plot(np.ones(N)*m2)
plt.plot(np.ones(N)*m3)
plt.xscale('log')
plt.show()
for a in actions:
print(a.mean)
return cumulative_average
```

3. Let us run the experiment with the following values:

```
c_1 = run_experiment(1.0, 2.0, 3.0, 0.1, 100000)
```

Let us look at *p versus x* value as shown in the *figure 8.4*:

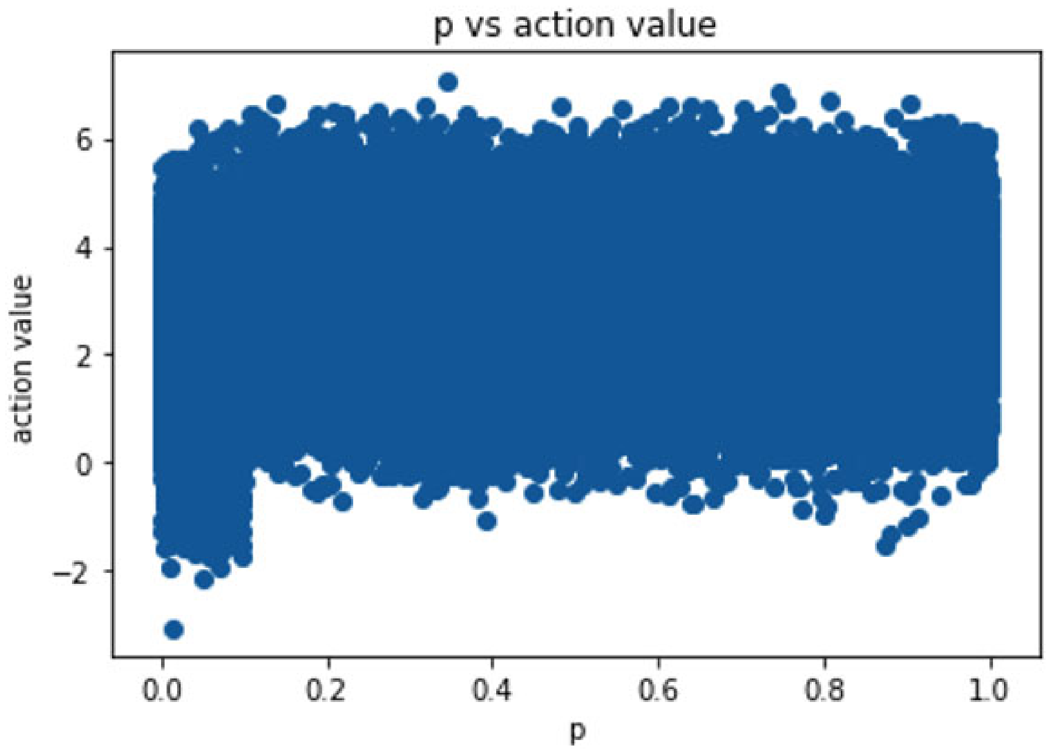


Figure 8.4: *Action value versus p*

As can be seen, the action value varies between *-2* and *6,* and *p* varies between *0* and *1* as shown in the *figure 8.5*:

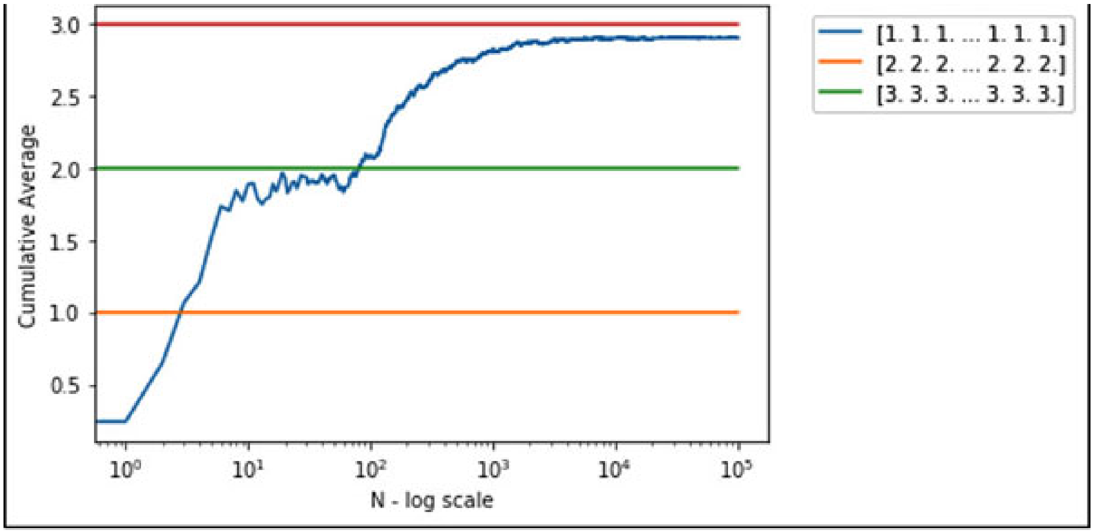


Figure 8.5: *Cumulate average of reward versus the episode number N*

Our algorithm tries to converge cumulative average towards **3,** which is the action with highest value.

Monotonic ϵ greedy policy improvement

For any -greedy policy , the -greedy policy with respect to is a monotonic improvement :

$$Q^{\pi}i\left(s,\pi_{i+1}(s)\right)=\sum_{a\in A}\pi_{i+1}(a|s)\,Q^{\pi_i}(s,a)$$

$$Q^{\pi}i\left(s,\pi_{i+1}(s)\right)=(\varepsilon/|A|)\sum_{a}Q^{\pi_i}(s,a)+(1-\varepsilon)\max_{a}Q^{\pi_i}(s,a)$$

Where $(\varepsilon/|A|)\sum_{a}Q^{\pi_i}(s,a)$ represents random with probability and ϵ, $(1-\varepsilon)\max_{a}Q^{\pi_i}(s,a)$represents greedy policy:

$$=\frac{1\varepsilon 1}{|A|}\sum_{a}Q^{\pi_i}\left(s_1,a\right)+1-\varepsilon_{max}Q^{a\in A}(s,a)\,\frac{1-\varepsilon}{1-\varepsilon}$$

$$=\quad+(1-\varepsilon)\,m_{a}^{\pi_i}Q^{\pi i}(s,a)\left[\frac{\sum_{a}\pi_i(a/s)-\varepsilon}{1-\varepsilon}\right]$$

$$=\quad\frac{(l-\varepsilon)}{l-\varepsilon}\sum_{a}\left(\pi_i(a|s)-\varepsilon\right)Q^{\pi i}(s,a)$$

Putting it all together:

$$= \frac{|\varepsilon|}{|A|} \sum_{a} Q^{\pi j}(s,a) + \sum_{a} \pi_{i}(a|s)\, Q^{\pi i}(s,a) - \frac{\varepsilon}{|A|} \sum Q^{\pi i}(s,a$$

$$= \sum_{a} \pi_{i}(a|s)\, Q^{\pi i}(s,a) = V^{\pi_i}$$

Before moving towards more advanced algorithms, let us look at TF-agents and the RL library from TensorFlow.

Let us look at the core papers and concepts which form the base for deep learning applied to reinforcement learning.

DQN networks

Controlling agents through high dimensional sensory inputs like vision and speech is a challenge in reinforcement learning. Most of the RL applications in the past relied on hand crafted features. Recent advances in deep learning have made it possible to extract high level features from sensory data of vision and speech. They rely on complex neural network architectures like CNN, perceptron's, restricted Boltzmann machines, and recurrent neural networks using supervised and unsupervised learning. Applying deep learning to reinforcement learning requires a large amount of hand labelled data.

RL applications must be able to learn from sparse scalar data. Deep learning data assumes static distribution of data whereas in reinforcement learning, data distribution changes as an algorithm learns new behavior.

DQN paper demonstrates that a convolutional neural network can overcome challenges listed earlier to learn successful control policies from raw video data in complex reinforcement learning environments. The network is trained on a variant of the *Q-learning algorithm*, with stochastic gradient descent to update the weights.

Modelling the reinforcement learning

Authors consider tasks in which an agent interacts with an environment E, in the case of an *Atari* emulator, in a sequence of actions, observations, and rewards. At every time-step, the agent selects an action from the set of legal game actions, $A = \{1, , K\}$. The action is passed to the emulator. It modifies its internal state and game score. *E* may be stochastic in distribution. The agent does not observe internal state of the emulator; instead, it observes an image $x_t = R_d$ from the emulator, which is a vector of raw pixel values representing the current screen. In addition, it receives a reward r_t, which is the change in game score. The game score depends on the whole prior sequence of actions and observations; feedback about an action is received after many thousands of time-steps have elapsed.

Neural networks are used to approximate the following function:

$$Q^*(s,a) = \max_{\pi} \mathrm{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi\right]$$

Where rt refers to rewards, which is the maximum sum of rewards r_t discounted by γ at each timestamp *t*, achievable by a behavior policy π after making an **observation** (**s**) and taking an **action** (**a**).

Reinforcement learning is unstable or can even diverge when a non-linear function approximator like a neural network is used to represent the action-value (also called *Q*) function. This instability has several causes. First one is that the correlations present in the sequence of observations, small updates to *Q* may significantly change the policy - *pie* and hence change the data distribution and also the correlations between the action-values (*Q*) and the target values *r*. Authors address these instabilities with a novel variant of *Q-learning*, which uses two key ideas:

- First is to use a biologically inspired mechanism termed as experience replay that randomizes over the data, hence removing correlations in the observation sequence and smoothing changes in the data distribution.
- Second is to iteratively update and adjust the action-values (*Q*) towards target values that are periodically updated, thereby, reducing correlations with the target. In the sections later, we will look at how to use DQN agent implemented in TF-agent library.

Atari game with deep reinforcement learning - DQN

This paper uses **OFF-policy** / **temporal difference** / **model free** for training a model which generates action from the state.

It consists of the following three ideas:

- **Idea 1**: Approximate Q-function using neural network:

```
def create_model(self):
  model = tf.keras.Sequential([
   Input((self.state_dim,)),
   Dense(32, activation='relu'),
   Dense(16, activation='relu'),
   Dense(self.action_dim)
  ])
  model.compile(loss='mse', optimizer=Adam(args.lr))
  return model
```

- **Idea 2**: Use **`target`** network:
  `self`.target_model = ActionStateModel(`self`.state_dim, self.action_dim)
- **Idea 3**: Use **`ReplayBuffer`** to increase data efficiency:

```
class ReplayBuffer:
   def __init__(self, capacity=10000):
    self.buffer = deque(maxlen=capacity)
   def put(self, state, action, reward, next_state, done):
    self.buffer.append([state, action, reward, next_state,
    done])
   def sample(self):
    sample = random.sample(self.buffer, args.batch_size)
    states, actions, rewards, next_states,
done = map(np.asarray, zip(*sample))
    states = np.array(states).reshape(args.batch_size, -1)
    next_states = np.array(next_states).reshape(args.batch_size,
    -1)
    return states, actions, rewards, next_states, done
   def size(self):
    return len(self.buffer)
```

Let us look at the full code listing:

1. Start with importing all the packages relevant to this sample:

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam
import gym
import argparse
import numpy as np
from collections import deque
import random
tf.keras.backend.set_floatx('float64')
```

2. Create a replay buffer to store the state, action reward, and next state:

```
class ReplayBuffer:
   def __init__(self, capacity=10000):
    self.buffer = deque(maxlen=capacity)
    self.batch_size = 32
   def put(self, state, action, reward, next_state, done):
    self.buffer.append([state, action, reward, next_state,
    done])
   def sample(self):
    sample = random.sample(self.buffer, self.batch_size)
    states, actions, rewards, next_states, done
       = map(np.asarray, zip(*sample))
    states = np.array(states).reshape(self.batch_size, -1)
    next_states = np.array(next_states).reshape(
       self.batch_size, -1)
    return states, actions, rewards, next_states, done
   def size(self):
    return len(self.buffer)
```

3. Next, create a **`ActionStateModel`** which is the *Q-model* approximation. It implements the model to predict the next action, given a current state. The model is a sequential model with four layers:

```
tf.keras.Sequential([
       Input((self.state_dim,)),
       Dense(32, activation='relu'),
       Dense(16, activation='relu'),
       Dense(self.action_dim)
      ])
```

4. The model is trained and used for prediction in **`get_action(self, state)`** method:

```
class ActionStateModel:
   def __init__(self, state_dim, action_dim):
    self.state_dim = state_dim
    self.action_dim = aciton_dim
    self.epsilon = 1.0 #args.eps
    self.lr = 0.005
    self.eps_min = 0.01
    self.eps_decay = 0.995
    self.model = self.create_model()
   def create_model(self):
    model = tf.keras.Sequential([
       Input((self.state_dim,)),
       Dense(32, activation='relu'),
       Dense(16, activation='relu'),
       Dense(self.action_dim)
    ])
    model.compile(loss='mse', optimizer=Adam(self.lr))
    return model
   def predict(self, state):
    return self.model.predict(state)
   def get_action(self, state):
    state = np.reshape(state, [1, self.state_dim])
    self.epsilon *= self.eps_decay
    self.epsilon = max(self.epsilon, self.eps_min)
    q_value = self.predict(state)[0]
    if np.random.random() < self.epsilon:
       return random.randint(0, self.action_dim-1)
    return np.argmax(q_value)
   def train(self, states, targets):
    self.model.fit(states, targets, epochs=1, verbose=0)
```

5. **`Agent`** class is the one which triggers the training and takes the **Gym** environment. We initialize the agent with the environment, state, and action dimensions. The two models are **`model`** and **`target_model`**:

```
class Agent:
    def __init__(self, env):
      self.env = env
      self.state_dim = self.env.observation_space.shape[0]
      self.action_dim = self.env.action_space.n
      self.model = ActionStateModel(self.state_dim,
      self.action_dim)
      self.target_model = ActionStateModel(self.state_dim,
  self.action_dim)
      self.target_update()
      self.batch_size = 32
      self.gamma = 0.95
      self.buffer = ReplayBuffer()
    def target_update(self):
      weights = self.model.model.get_weights()
      self.target_model.model.set_weights(weights)
    def replay(self):
      for _in range(10):
        states, actions, rewards, next_states, done
          = self.buffer.sample()
          targets = self.target_model.predict(states)
          next_q_values = self.target_model.predict(
          next_states).max(axis=1)
            targets[range(self.batch_size), actions]
          = rewards + (1-done) * next_q_values * self.gamma
          self.model.train(states, targets)
    def train(self, max_episodes=1000):
      for ep in range(max_episodes):
```

```
      done, total_reward = False, 0
      state = self.env.reset()
      while not done:
     action = self.model.get_action(state)
     next_state, reward, done, _ = self.env.step(action)
     self.buffer.put(state, action, reward*0.01,
next_state, done)
     total_reward += reward
     state = next_state
      if self.buffer.size() >= self.batch_size:
     self.replay()
      self.target_update()
      print('EP{} EpisodeReward={}'.format(ep, total_ reward))
```

Main method is **`train(..)`** which calls the model to get action and update the buffer. Replay uses two models:

- **`target_models`** is used to predict targets.
- **`next_q_values`** which are then called to train the model.

6. Let us instantiate the agent with **`CartPole`** environment and run it for **`100`** episodes:

```
env = gym.make('CartPole-v1')
agent = Agent(env)
agent.train(max_episodes=100)
```

Output

```
EP0
EpisodeReward=21.0
EP1 EpisodeReward=26.0
EP2 EpisodeReward=13.0
EP3 EpisodeReward=23.0
...
EP97 EpisodeReward=137.0
EP98 EpisodeReward=200.0
EP99 EpisodeReward=187.0
```

In the next section, we will look at dueling DQN which is an advanced form of DQN network.

The reference paper is as follows:

Playing Atari with Deep Reinforcement Learning (**https://arxiv.org/abs/1312.5602**).

Double DQN network

We have learnt that parameters for the network to find the *Q* function are:

$$\theta_{t+1} = \theta_t + \alpha\left(Y_t^Q - Q\left(S_t, A_t; \theta_t \right) \right) \nabla_{\theta_t} Q\left(S_t, A_t; \theta_t \right)$$

where α (alpha) is a scalar step size and the target Y_t^Q is defined as follows:

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max Q\left(S_{t+1}, a; \theta_t \right)$$

The **`max`** operator in standard *Q-learning* and DQN in the preceding equations uses the same values both to select and evaluate an action. This makes it more likely to select overestimated values, resulting in over-optimistic value estimates. To prevent this, we can decouple the selection from the evaluation. This is the idea behind *double Q-learning*.

In the *double Q-learning algorithm*, two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, θ and θ'. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. Let us rewrite the preceding equation by replacing a with the parameterized value function $\operatorname{argmax} Q\left(s_{t+1}, a; \theta_t \right)$

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q\left(S_{t+1}, \mathrm{arg}max Q\left(S_{t+1}, a; \theta_t \right); \theta_t \right)$$

Then, the double DQN network can be written as:

$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \mathrm{arg}max Q\left(S_{t+1}, a; \theta_t \right); \theta'_t \right)$$

The selection of the action in the **`argmax`** is still using the online weights θ_t. This means that, as in *Q learning*, we are estimating the value of the greedy policy according to the current values, as defined by θ_t. A second set of weights θ'_t is used to evaluate the value of this policy. The second set of weights can be updated symmetrically by switching the roles of θ and θ'.

Implementation of double DQN using TensorFlow

Most of the implementations are the same as the one we implemented on DQN:

1. Create a replay buffer to the store state, action reward, and next state:

```
class ReplayBuffer:
  def __init__(self, capacity=10000):
    self.buffer = deque(maxlen=capacity)
    self.batch_size = 32

  def put(self, state, action, reward, next_state, done):
    self.buffer.append([state, action, reward, next_state,
    done])

  def sample(self):
    sample = random.sample(self.buffer, self.batch_size)
    states, actions, rewards, next_states, done
      = map(np.asarray, zip(*sample))
    states = np.array(states).reshape(self.batch_size, -1)
    next_states = np.array(next_states).reshape(
      self.batch_size, -1)
    return states, actions, rewards, next_states, done
  def size(self):
    return len(self.buffer)
```

2. Next, create an **ActionStateModel** which is the *Q-model* approximation. It implements the model to predict the next action given a current state. The model is a sequential model with four layers:

```
tf.keras.Sequential([
      Input((self.state_dim,)),
      Dense(32, activation='relu'),
      Dense(16, activation='relu'),
      Dense(self.action_dim)
     ])
```

3. The model is trained and used for prediction in **get_action(self, state)** method:

```
class ActionStateModel:
  def __init__(self, state_dim, action_dim):
    self.state_dim = state_dim
    self.action_dim = action_dim
    self.epsilon = 1.0 #args.eps
    self.lr = 0.005
    self.eps_min = 0.01
    self.eps_decay = 0.995
    self.model = self.create_model()
  def create_model(self):
    model = tf.keras.Sequential([
      Input((self.state_dim,)),
      Dense(32, activation='relu'),
      Dense(16, activation='relu'),
      Dense(self.action_dim)
     ])
    model.compile(loss='mse', optimizer=Adam(self.lr))
    return model
   def predict(self, state):
    return self.model.predict(state)
   def get_action(self, state):
    state = np.reshape(state, [1, self.state_dim])
    self.epsilon *= self.eps_decay
    self.epsilon = max(self.epsilon, self.eps_min)
    q_value = self.predict(state)[0]
    if np.random.random() < self.epsilon:
      return random.randint(0, self.action_dim-1)
    return np.argmax(q_value)
   def train(self, states, targets):
    self.model.fit(states, targets, epochs=1, verbose=0)
```

4. **`Agent`** class is the one which triggers the training and takes *Gym* environment. We initialize the **`Agent`** with the environment, state, and action dimensions. The two models are **`model`** and **`target_model`**:

```
  class Agent:
    def __init__(self, env):
     self.env = env
     self.state_dim = self.env.observation_space.shape[0]
     self.action_dim = self.env.action_space.n
     self.model = ActionStateModel(self.>state_dim,
     self.action_dim)
     self.target_model = ActionStateModel(self.state_dim,
   self.action_dim)
     self.target_update()
     self.batch_size = 32
     self.gamma = 0.95
     self.buffer = ReplayBuffer()
    def target_update(self):
     weights = self.model.model.get_weights()
     self.target_model.model.set_weights(weights)
    def replay(self):
     for_ in range(10):
     states, actions, rewards, next_states, done
    = self.buffer.sample()
            targets = self.target_model.predict(states)
            next_q_values = self.target_model.predict(next_states)[
    range(self.batch_size),
            np.argmax(self.model.predict(next_states), axis=1)]
            targets[range(self.batch_size), actions]
    = rewards + (1-done) * next_q_values * self.gamma
            self.model.train(states, targets)
      def train(self, max_episodes=1000):
        for ep in range(max_episodes):
            done, total_reward = False, 0
            state = self.env.reset()
            while not done:
            action = self.model.get_action(state)
            next_state, reward, done, _ = self.env. step(action)
            self.buffer.put(state, action, reward*0.01,
  next_state, done)
            total_reward += reward
            state = next_state
            if self.buffer.size() >= self.batch_size:
            self.replay()
            self.target_update()
            print('EP{} EpisodeReward={}'.format(ep, total_ reward))
```

5. Notice in replay that the code in bold is the one which has changed:

```
# next_q_values = self.target_model.predict(
# next_states).max(axis=1)
next_q_values char-style-override-107
self.target_model.predict(next_states)[
range(self.batch_size),
        np.argmax(self.model.predict(next_states), axis=1)]
```

6. Let us run the agent:

```
env = gym.make('CartPole-v1')
agent = Agent(env)
agent.train(max_episodes=100)
EP96 EpisodeReward=484.0
EP97 EpisodeReward=209.0
EP98 EpisodeReward=349.0
EP99 EpisodeReward=327.0
```

In the next section, we look at the evolution of *actor critic network*.

Value-based versus policy-based RL

Value-based RL has a learnt value function but implicit policy, for example, -greedy. Policy-based RL has no value function but learnt policy. Actor critic has learnt value and policy. *Figure 8.7* shows value-based versus policy-based reinforcement learning techniques:

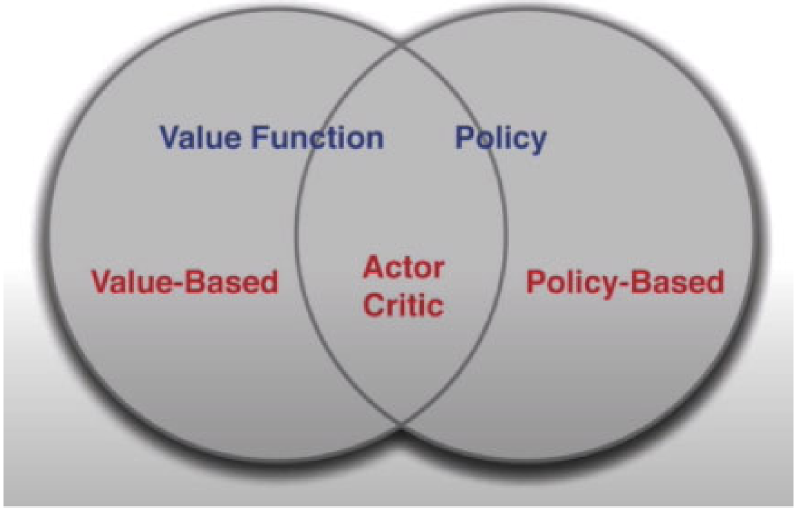


Figure 8.7: *Value based versus policy based RL*

Next, let us look at the actor critic network which is at the intersection of value-based and policy-based RL.

Actor critic network

Actor-only methods use parameterized family of policies. The gradient of the performance with respect to the actor parameters is estimated by simulation. The parameters are updated in the direction of improvement. A potential drawback of such methods is that the gradient estimators have a larger variance.

Critic-only methods rely only on value function approximation and aim at learning an approximate solution to the *Bellman equation*, which then hopefully prescribes a near-optimal policy.

Actor-critic methods combine the strong points of **actor-only** and **critic only** methods. The critic uses an approximation architecture and simulation to learn a value function, which is used to update the actor's policy parameters Actor critic algorithms combine the strong points of **actor-only** and **critic-only** methods. Critic based methods used approximation and simulation to help come up with a value function which is then used to improve the actor. Such methods, if they are gradient based, potentially have desirable convergence properties, in contrast to critic-only methods where convergence is guaranteed in very limited settings. These algorithms hold the promise of delivering faster convergence (due to variance reduction), when compared to *actor-only* methods:

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
import gym
import argparse
import numpy as np
gamma = 0.99
update_interval = 5
actor_lr = 0.0005
critic_lr = 0.001
```

1. Create **`Actor`** class which encapsulates a fully connected model, optimizer and state, and action dimensions. It implements the following methods:
   - **`create_model(..)`**
   - **`compute_loss(..)`**
   - **`train(..)`**
2. Model takes **`state_dim`** and outputs **`action_dim`**:

```
class Actor:
  def __init__(self, state_dim, action_dim):
    self.state_dim = state_dim
    self.action_dim = action_dim
    self.model = self.create_model()
    self.opt = tf.keras.optimizers.Adam(actor_lr)
  def create_model(self):
    return tf.keras.Sequential([
      Input((self.state_dim,)),
      Dense(32, activation='relu'),
      Dense(16, activation='relu'),
      Dense(self.action_dim, activation='softmax')
])
  def compute_loss(self, actions, logits, advantages):
    ce_loss = tf.keras.losses.SparseCategoricalCrossentropy(
      from_logits=True)
    actions = tf.cast(actions, tf.int32)
    policy_loss = ce_loss(
     actions, logits,
      sample_weight=tf.stop_gradient(advantages))
    return policy_loss
  def train(self, states, actions, advantages):
    with tf.GradientTape() as tape:
      logits =self.model(states, training=True)
      loss =self.compute_loss(
     actions, logits, advantages)
    grads = tape.gradient(loss, self.model.trainable_ variables)
    self.opt.apply_gradients(zip(grads,
      self.model.trainable_variables))
    return loss
```

3. **`Critic`** class is implemented which encapsulates a fully connected model with four layers. The last layer has a single neuron:

```
class Critic:
  def __init__(self, state_dim):
    self.state_dim = state_dim
    self.model = self.create_model()
    self.opt = tf.keras.optimizers.Adam(critic_lr)
  def create_model(self):
    return tf.keras.Sequential([
      Input((self.state_dim,)),
      Dense(32, activation='relu'),
      Dense(16, activation='relu'),
      Dense(16, activation='relu'),
      Dense(1, activation='linear')
     ])
  def compute_loss(self, v_pred, td_targets):
    mse = tf.keras.losses.MeanSquaredError()
    return mse(td_targets, v_pred)
  def train(self, states, td_targets):
    with tf.GradientTape() as tape:
      v_pred = self.model(states, training=True)
      assert v_pred.shape == td_targets.shape
      loss = self.compute_loss(v_pred,
    tf.stop_gradient(td_targets))
    grads = tape.gradient(loss, self.model.trainable_variables)
    self.opt.apply_gradients(zip(grads,
     self.model.trainable_variables))
    return loss
```

4. Bring it all together with the **`Agent`**. It captures the *Gym* environment, actor critic, and the rewards. The **`train`** method steps are described as follows:

```
class Agent:
  def __init__(self, env):
    self.env = env
    self.state_dim = self.env.observation_space.shape[0]
    self.action_dim = self.env.action_space.n
    self.actor = Actor(self.state_dim, self.action_dim)
    self.critic = Critic(self.state_dim)
    self.rewards = []
  def td_target(self, reward, next_state, done):
    if done:
      return reward
    v_value = self.critic.model.predict(
      np.reshape(next_state, [1, self.state_dim]))
    return np.reshape(reward + gamma * v_value[0], [1, 1])
  def advantage(self, td_targets, baselines):
    return td_targets - baselines
  def list_to_batch(self, list):
    batch = list[0]
    for elem in list[1:]:
      batch = np.append(batch, elem, axis=0)
    return batch
  def train(self, max_episodes=1000):
    for ep in range(max_episodes):
      state_batch = []
      action_batch = []
      td_target_batch = []
      advatnage_batch = []
      episode_reward, done = 0, False
      state = self.env.reset()
      while not done:
      # self.env.render()
      probs = self.actor.model.predict(
      np.reshape(state, [1, self.state_dim]))
      action = np.random.choice(self.action_dim, p=probs[0])
      next_state, reward, done, _ = self.env. step(action)
      state = np.reshape(state, [1, self.state_dim])
      action = np.reshape(action, [1, 1])
      next_state = np.reshape(next_state,
 [1, self.state_dim])
      reward = np.reshape(reward, [1, 1])
      td_target = self.td_target(reward * 0.01, next_state,
done)
      advantage = self.advantage(
      td_target, self.critic.model.predict(state))
      state_batch.append(state)
      action_batch.append(action)
      td_target_batch.append(td_target)
      advatnage_batch.appendf(advantage)
      if len(state_batch) >= update_interval or done:
      states = self.list_to_batch(state_batch)
      actions = self.list_to_batch(action_batch)
      td_targets = self.list_to_batch(td_target_batch)
      advantages = self.list_to_batch(advatnage_batch)
      actor_loss = self.actor.train(states, actions,
advantages)
      critic_loss = self.critic.train(states, td_ targets)
      state_batch = []
      action_batch = []
      td_target_batch = []
      advatnage_batch = []
      episode_reward += reward[0][0]
      self.rewards.append(episode_reward)
      state = next_state[0]
      print('EP{} EpisodeReward={}'.format(ep, episode_reward))
```

5. Running the agent:

```
env_name = 'CartPole-v1'
env = gym.make(env_name)
agent = Agent(env)
agent.train()
```

Output will take some time to appear:

```
EP0 EpisodeReward=14.0
EP1 EpisodeReward=23.0
EP2 EpisodeReward=17.0
…
EP997 EpisodeReward=226.0
EP998 EpisodeReward=188.0
EP999 EpisodeReward=187.0
```

You will notice from the following graph that the rewards maximize to `500`:

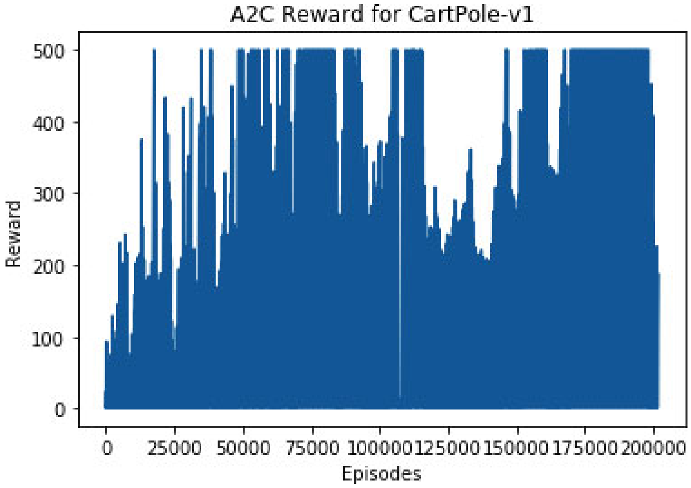


Figure 8.8: *Actor critic reward versus episodes plot*

Till now, we focused on the basics of RL. Now, let us look at a new library introduced by TensorFlow team.

Introducing TF-agents

TF-agents (**https://github.com/tensorflow/agents**) make implementing, deploying, and testing RL algorithms easier. It provides well-tested and modular components that can be modified and extended. It enables faster code iteration with good test integration and benchmarking.

To get started, it provides *Colab* based tutorials. We will start with DQN tutorial to get an agent up and running in the *CartPole* environment.

TF-agents are under active development and the interfaces change frequently. It can be found at the following link:

https://github.com/tensorflow/agents

TF-agent

Reinforcement learning library written on top of TensorFlow makes experimentation easier and the following components are implemented out of the box. *Figure 8.9* illustrates basic components of the TF-agent implementation:

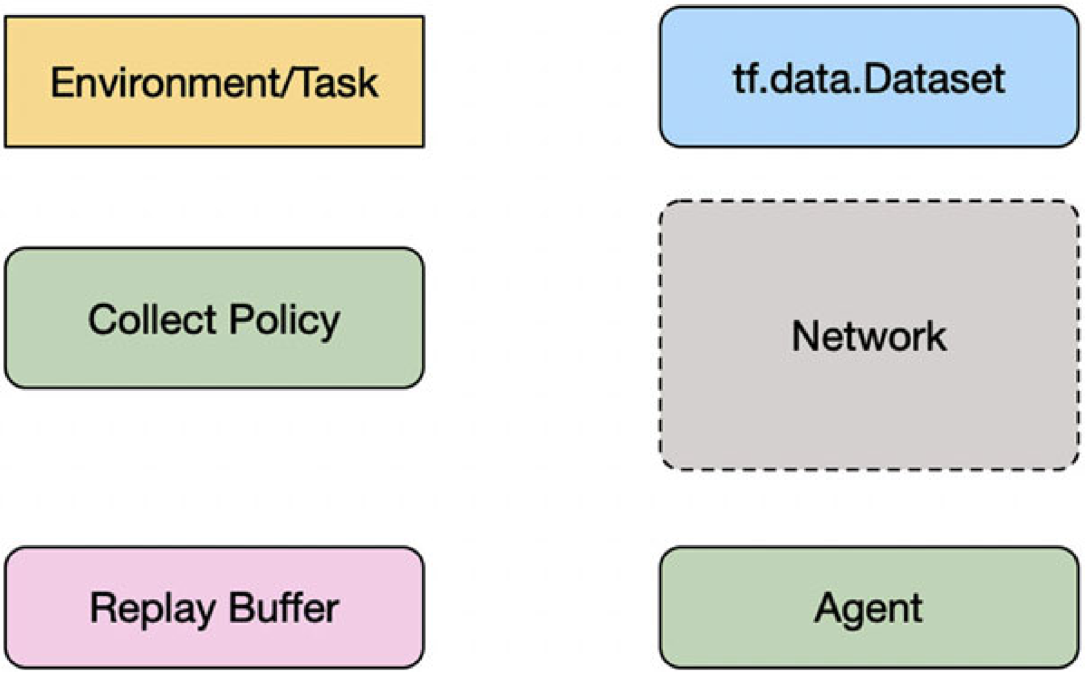


Figure 8.9: *Components of tf-agent library*

Let us look at how some of the basic things can be used using TF agents. Actor and critic are two key players in reinforcement learning. Using TF agents, these can be implemented using the following code snippets:

- **`ActorDistributionNetwork`**: This creates an actor producing either normal or categorical distribution. It inherits from **`DistributionNetwork`** (**https://www.tensorflow.org/agents/api_docs/python/tf_agents/networks/network/DistributionNetwork**), **`Network`** (**https://www.tensorflow.org/agents/api_docs/python/tf_agents/networks/Network**). Network is a class used to represent networks used by TF-agents' policies and agents:

  ```
  actor_net = actor_distribution_network.ActorDistributionNetwork(
  train_env.observation_spec(), train_env.action_spec(),
  fc_layer_params=fc_layer_params)
  ```

- **`CriticNetwork`**: This creates a critic network, inherited from the **`Network`** class explained earlier:

  ```
  tf_agents.agents.ddpg.critic_network.CriticNetwork(
  input_tensor_spec, .., name='CriticNetwork')
  ```

- **`SacAgent`**: **Soft Actor Critic** (**SAC**) is an algorithm that optimizes a stochastic policy in an off-policy way (An off-policy is independent of the agent's actions. It figures out the optimal policy regardless of the agent's motivation.), forming a bridge between stochastic policy optimization and **Deep Deterministic Policy Gradient** (**DDPG**)-style approaches. It is not a direct successor to twin delayed DDPG-TD3. It incorporates the clipped *double-Q trick* and due to the inherent stochasticity (the quality of lacking any predictable order or plan) of the policy in SAC, it also helps benefiting from target policy smoothing.

An important feature of SAC is **entropy regularization**. The policy is trained for maximizing a trade-off between expected return and entropy, a measure of randomness in the policy (**https://en.wikipedia.org/wiki/Entropy_(information_theory)**). It has a close connection to the exploration-exploitation trade-off. In this trade-off, increasing entropy results in more exploration can accelerate learning subsequently. SAC can also prevent the policy from prematurely converging to a local optimum.

SAC in TensorFlow agents is implemented using the following method:

```
tf_agents.agents.SacAgent(
   time_step_spec: tf_agents.trajectories.time_step.TimeStep,
   action_spec: tf_agents.typing.types.NestedTensorSpec,
   critic_network: tf_agents.networks.Network, actor_network:
   tf_agents.networks.Network,
…)
```

Next, let us look at replay buffers.

Replay buffers

Reinforcement learning algorithms use replay buffers to store trajectories of experience while executing a policy in an environment. Replay buffers can be queried for a subset of trajectories (either a sequential subset or a sample) to *replay* the agent's experience during training.

In this book, we will explore two types of replay buffers: **Python-backed** and **TensorFlow-backed**, sharing a common API.

We describe the API and implementation, each of the buffer implementations, and see how to use them during data collection training.

`TFUniformReplayBuffer` is the most commonly used replay buffer in TF-agents. Hence, we are using it in our example here. In **`TFUniformReplayBuffer`**, the backing buffer storage is done by TensorFlow variables and hence, is a part of the compute graph.

The buffer stores batches of elements and has a maximum capacity **`max_length`** elements per batch segment. Thus, the total buffer capacity is **`batch_size x max_length`** elements. The elements stored in the buffer must all have a matching data spec. When the replay buffer is used for data collection, the spec is the agent's collect data spec:

```
data_spec = (
    tf.TensorSpec([3], tf.float32, 'action'),
    (
      tf.TensorSpec([5], tf.float32, 'lidar'),
      tf.TensorSpec([3, 2], tf.float32, 'camera')
    )
)
batch_size = 32
max_length = 1000
replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
   data_spec,
   batch_size=batch_size,
   max_length=max_length)
```

- **Writing to the replay buffer**: Writing is done using the **`add_batch`** method:

  ```
  action = tf.constant(1 * np.ones(
     data_spec[0].shape.as_list(), dtype=np.float32))
     lidar = tf.constant(
     2 * np.ones(data_spec[1][0].shape.as_list(), dtype=np.float32))
  camera = tf.constant(
     3 * np.ones(data_spec[1][1].shape.as_list(), dtype=np.float32))
  values = (action, (lidar, camera))
  values_batched = tf.nest.map_structure(lambda t: tf.stack([t] *
  batch_size),
          values)
  replay_buffer.add_batch(values_batched)
  ```

  Next, let us look at reading from the buffer.

- **Reading from buffer**: Once we have written to the buffer, reading can be done in multiple ways:

  ```
  for _ in range(5):
    replay_buffer.add_batch(values_batched)
  sample = replay_buffer.get_next(sample_batch_size=10,
  num_steps=1)
  print(len(sample))
  ```

- **Convert the `replay` buffer**: Convert the **`replay`** buffer to a **`tf.data.Dataset`** and iterate through it:

  ```
  dataset = replay_buffer.as_dataset(
  sample_batch_size=4,
    num_steps=2)
  ```

```
iterator = iter(dataset)
print("Iterator trajectories:")
trajectories = []
for _ in range (3):
 t, _ = next(iterator)
 trajectories.append(t)
print(tf.nest.map_structure(lambda t: t.shape, trajectories))
```

- Read all elements in the **`replay`** buffer:

```
trajectories = replay_buffer.gather_all()
print("Trajectories from gather all:")
print(tf.nest.map_structure(lambda t: t.shape, trajectories))
```

Output will be:

```
2
Iterator trajectories:
[(TensorShape([4, 2, 3]), (TensorShape([4, 2, 5]),
TensorShape([4, 2, 3, 2]))), (TensorShape([4, 2, 3]),
(TensorShape([4, 2, 5]), TensorShape([4, 2, 3, 2]))),
(TensorShape([4, 2, 3]), (TensorShape([4, 2, 5]),
TensorShape([4, 2, 3, 2])))]
Trajectories from gather all:
(TensorShape([32, 46, 3]), (TensorShape([32, 46, 5]),
TensorShape([32, 46, 3, 2])))
```

Environments

The following environments are supported by the **`tf-agent`** library:

- Open AI gym
- Atari
- Mujoco
- PyBullet
- DM control

These are explained as follows:

- **Loading Open AI gym**

```
environment = suite_gym.load('CartPole-v0')
print('action_spec:', environment.action_spec())
print('time_step_spec.observation:',
environment.time_step_spec().observation)
print('time_step_spec.step_type:', environment.time_step_spec().step_type)
print('time_step_spec.discount:', environment.time_step_spec().discount)
print('time_step_spec.reward:', environment.time_step_spec().reward)
```

Output shows the variable values:

```
action_spec: BoundedArraySpec(shape=(), dtype=dtype('int64'),
name='action', minimum=0, maximum=1)
time_step_spec.observation: BoundedArraySpec(shape=(4,),
dtype=dtype('float32'), name='observation', minimum=
[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38],
maximum=[4.8000002e+00 3.4028235e+38 4.1887903e-01
3.4028235e+38])
time_step_spec.step_type: ArraySpec(shape=(),
dtype=dtype('int32'), name='step_type')
time_step_spec.discount: BoundedArraySpec(shape=(),
dtype=dtype('float32'), name='discount', minimum=0.0,
maximum=1.0)
time_step_spec.reward: ArraySpec(shape=(),
dtype=dtype('float32'), name='reward')
```

Similarly, we can load other environments like Atari.

- **Loading Atari**:

```
from tf_agents.environments import suite_atari
env_atari = suite_atari.load('Pong-v0')
env_atari.action_spec()
```

Let us look at the agents available:

The following agents are implemented in the **`tf-agents`** library:

- A DQN, or **Deep Q-Network**, approximates a state-value function in a *Q-Learning framework* with a neural network. In the *Atari Games case*, they take in several frames of the game as an input, and output state values for each action as an output.
- DDPG is used in continuous action setting. In DDPG, actor computes the action directly instead of a probability distribution over actions.

  Reference: *Continuous control with deep reinforcement learning - Lilicrap et al*
- A PPO agent implements the PPO algorithm from (*Schulman, 2017*):

  https://arxiv.org/abs/1707.06347 PPO is a simplification of the TRPO algorithm (described as follows), both these algorithms add stability to policy gradient RL, they allow multiple updates per batch of on-policy data, by limiting the KL divergence between the policy that sampled the data and the updated policy.

- TD3 is also supported. It extends DDPG by adding an extra critic network and using the minimum of the two critic values to reduce overestimation bias.
- TRPO enforces a hard optimization constraint, but is a complex algorithm, which makes it harder to use. PPO approximates the effect of TRPO by using a soft constraint. There are two methods presented in the paper for implementing the soft constraint:
    - an adaptive KL loss penalty
    - limiting the objective value based on a clipped version of the policy importance ratio. (This implementation handles both and allows the user to use either method.)
- A reinforce agent implements the reinforce algorithm from (*Williams, 1992*)

  https://www-anw.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf
- A soft actor-critic agent implements the **Soft Actor-Critic** (**SAC**) algorithm from *Soft Actor-Critic Algorithms and Applications* by *Haarnoja et al (2019).*

Q-learning is a basic form of reinforcement learning which uses *Q-values* (also called action values) to iteratively improve the behavior of the learning agent.

Q-values or action-values: Q-values are defined for states and actions.

Policy gradient methods are reinforcement learning techniques that rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent.

Mechanics of reinforcement learning

Let us look at the key components and their interaction in reinforcement learning. On the left side, we have collection aspects. We have a policy which is going to interact with the environment. We have replay buffer feeding from the collection into training to make the agent better.

Components of reinforcement learning

We have two main pipelines – **collection** and **training**. Collection pipeline collects experience from the environment and sends it to the replay buffers. These buffers are used for training dataset. Agent uses this dataset to improve the policy using neural network. These pipelines are illustrated in *figure 8.10*:

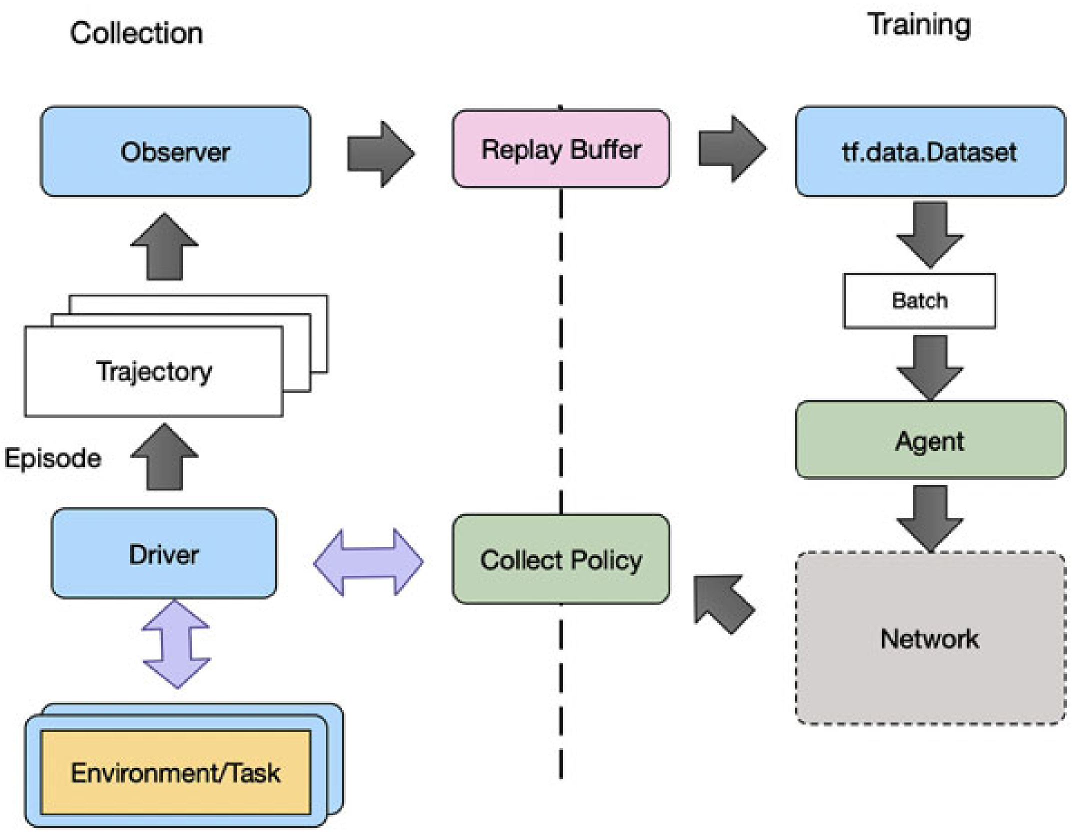


Figure 8.10: *Interaction between various components of tf-agents library*

Let us focus on the environment and its definition:

1. Environments can be created by extending **`py_environment.PyEnvironment`**. It has the following methods:

```
    def td_target(self, reward, next_state, done):
      if done:
        return reward
class BreakoutEnv(py_environment.PyEnvironment):
  def observation_spec(self):
      # defines the observations
      pass
  def action_spec(self):
      # defines the action
      pass
  def _reset(self):
      # reset the state
      pass
    def _apply_action(self, action):
      # apply action
      pass
    def _step(self, action):
      # apply action and return observation and reward
      observation, reward = self._apply_action(action)
      return TimeStep(observation, reward)
```

- **`_observation_spec(..)`**: defines the kind of observations this environment provides
- **`_action_spec(..)`**: actions supported
- **`_reset(..)`**: resetting the state of the breakout environment
- **`_apply_action(..)`**: apply action to the environment
- **`_step(..)`**: given an action, gives us a new observation and reward

2. Defining the policy and the environment:

```
env = BreakoutEnv(…)
policy =. MyPolicy(..)
time_step = env.reset()
episode_return = 0.0
while not time_step.is_last():
  policy_step = policy.action(time_step)
  time_step = env.step(policy_step.action)
  episode_return = += time_step.reward
```

Policies take observations and emit parameters of distribution of actions. In case of breakout, observations are images. The following *figure 8.11* illustrates the logical flow between various components:

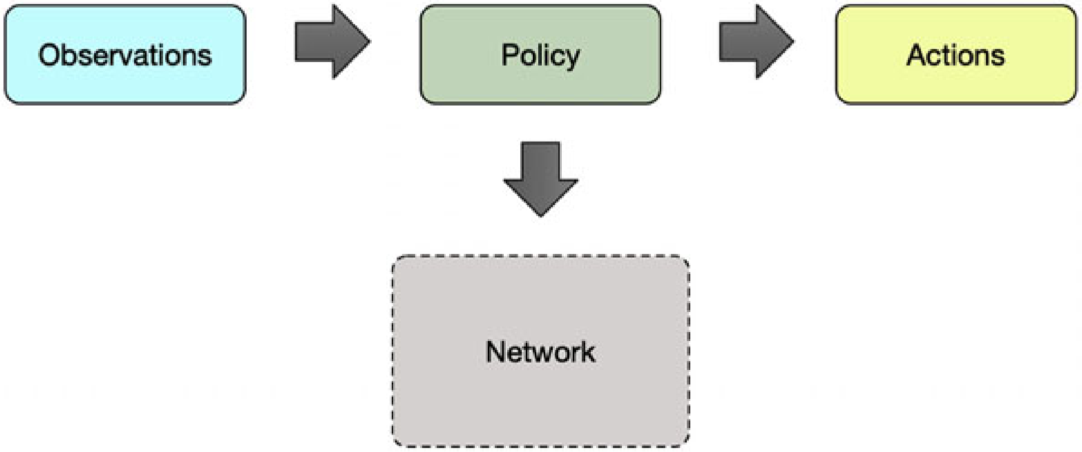


Figure 8.11: *Relation between observations, policy, and actions*

Let us define a custom network using the parent class of **`tf_agents.networks.Network`**.

3. Defining a custom network: Network used for deep *Q-learning* is essentially a container for Keras sequential network:

   ```
   class MyQNetwork(tf_agents.networks.Network):
      def __init__(self, no_of_actions):
       self._forward = tf.keras.Sequential(
      [tf.keras.layers.Conv2D(32,(8,8),4,activation='relu'),
       tf.keras.layers.Dense(number_of_actions)])
   ```

   Input goes through a couple of convolutional layers and finally emits the **`no_of_actions`**. **`call`** is the most important method in the preceding **`Network`** class. It takes observations and state, and emits the **`logits`**:

   ```
   def call(self, state=(), observations):
      logits = self._forward(observations)
       return logits, state
   ```

Policies

`tf-agent` comes with prebuilt policies, but you can also build your own policy. It takes a network for initialization. Fundamental method is **`_distribution(..)`**:

```
from tf_agents.policies import tf_policy
class MyPolicy(tf_policy.TFPolicy):
    def __init__(self, network):
     pass
    def _distribution(self, time_step, policy_state):
    logits, next_state = self._network(time_step.observation,
     policy_state, )
     return PolicyStep(tfp.distribution.Categorical(logits),
       next_state, info={"logits:",logits})
```

`_distribution(..)` method takes **`time_step`** and policy state as the input, passes it through the network, and emits the **`PolicyStep`**, which is a tuple of three things – **distribution of logits**, **`next_state`**, and **`logits`**.

Training pipeline

Agent class encompasses main RL algorithm. It reads batches of data (trajectories) and updating the neural network:

1. To create an agent, there are standard implementations. It takes **`q_network`** and optimizer:

   ```
   agent = DqnAgent(q_network=.., optimizer=...)
   ```

2. Then, we collect policy from the agent:

   ```
   policy = agent.collect_policy()
   ```

3. The main method on the agent is **`train`**. Here, we train from a batch of experience:

   ```
   loss_info = agent.train(trajectories = projection)
   ```

4. Create a network:

   ```
   q_net = q_network.QNetwork(observation_spec, action_spec)
   ```

   Figure 8.12 shows the steps involved in training the network by capturing the dataset from observations and feeding it back into the pipeline:

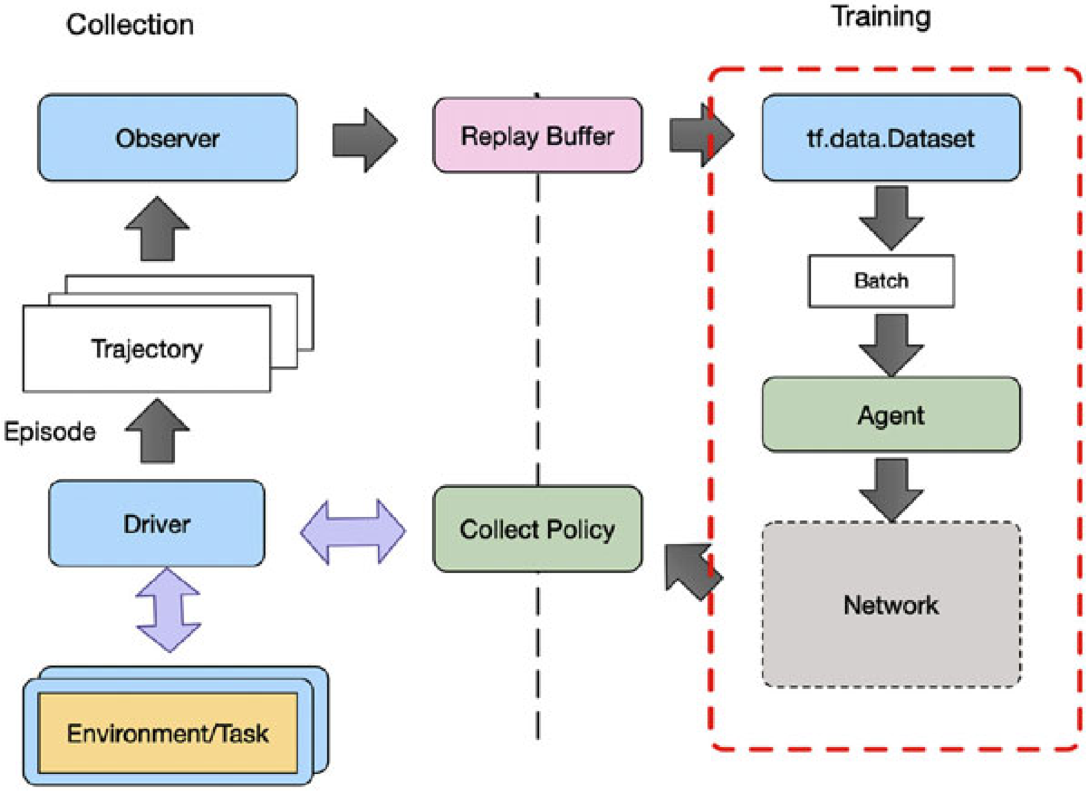


Figure 8.12: *Training the underlying network from collected data*

5. Build an agent with that network:

   ```
   dqn_agent = DqnAgent(
     q_network = q_net,
     optimizer = AdamOptimizer(learning_rate = learning_rate))
   ```

6. Get experience from **`tf.data.Dataset`** and train the agent:

   ```
   dataset = replay_buffer.as_dataset().step(num_steps = 2).fetch(3
   for batched_experience in dataset:
     agent.train(batched_experience)
   ```

Next, we look at the collect experience which feeds into the **`replay`** buffer.

Collect experience

1. First, create a TensorFlow environment, create a **`replay`** buffer, and then the driver:

   ```
   tf_env = TFPyEnvironment([..])
   replace_buffer = TFUniformReplayBuffer(agent.collect_dataset)
   driver = DynmaicStepDriver(tf_env, agent.collect_policy,
   observers=[replay_buffer.add_batch], num_step=100)
   ```

2. The driver accepts environment, **`collect_policy`** and a number of call backs (observers). When you call **`driver.run()`**, it populates the **`replay_buffer`** with *100* more steps.

   The following figure summarizes the **collect** and **train** flow with actual classes and method:

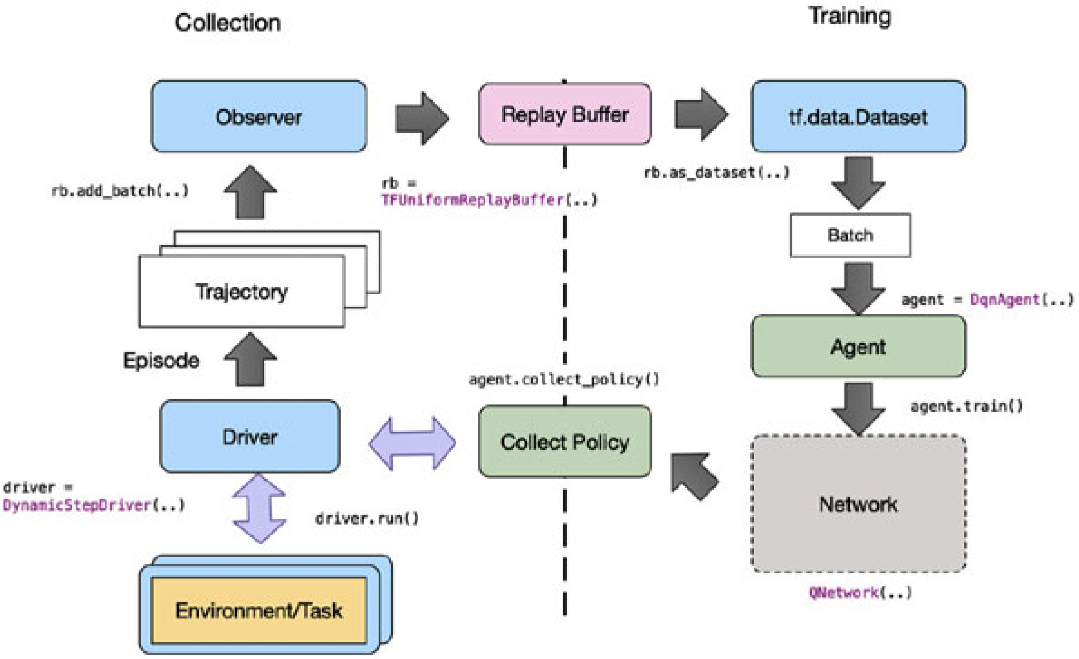


Figure 8.13: *Collect and train sequence with implementation classes and methods*

We have looked at the mechanics of collection and training pipelines. Let us look at an SQN agent using TF-agents in the next section.

Simple DQN agent

Let us build a very simple DQN agent with two observations and actions:

1. Create a dummy network that extends **`network.Network`**. It has a dummy dense layer:

   ```
   class DummyNet(network.Network):
   ```

```
  def __init__(self,
       observation_spec,
       action_spec,
       l2_regularization_weight=0.0,
       name=None):
   super(DummyNet, self).__init__(
    observation_spec, state_spec=(), name=name)
   num_actions = action_spec.maximum - action_spec.minimum + 1
   # Store custom layers that can be serialized through the
   Checkpointable API.
   self._dummy_layers = [
    tf.keras.layers.Dense(
      num_actions,
      kernel_regularizer=tf.keras.regularizers.l2(
      l2_regularization_weight),
      kernel_initializer=tf.constant_initializer([[num_ actions,
      1],
        [1, 1]]),
      bias_initializer=tf.constant_initializer([[1], [1]]))
     ]
  def call(self, inputs, step_type=None, network_state=()):
   del step_type
   inputs = tf.cast(inputs, tf.float32)
   for layer in self._dummy_layers:
    inputs = layer(inputs)
   return inputs, network_state
```

2. Create a base class. It inherits from **`tf.test.TestCase`** and **`parameterized.TestCase`** some infrastructure elements:

```
 class DqnAgentBase(tf.test.TestCase, parameterized.TestCase):
  def setUp(self):
   super(DqnAgentBase, self).setUp()
   self._observation_spec = tensor_spec.TensorSpec([2],
   tf.float32)
   self._time_step_spec =
   ts.time_step_spec(self._observation_spec)
   self._action_spec = tensor_spec.BoundedTensorSpec((),
   tf.int32, 0, 1)
```

3. Next, we implement the implementation class. Using the kernel initializer *[[2, 1], [1, 1]]* and bias initializer *[[1], [1]]* from *DummyNet*, we can calculate the following values:

```
Q-value for first observation/action pair: 2 * 1 + 1 * 2 + 1 = 5
Q-value for second observation/action pair: 1 * 3 + 1 * 4 + 1 =
8
(Here we use the second row of the kernel initializer above,
since the
chosen action is now 1 instead of 0.)
```

For the target *Q-values* here, note that since we've replaced *5* and *7* with *-5* and *-7*, it is better to use action *1* with a kernel of *[1, 1]* instead of action *0* with a kernel of *[2, 1]*:

```
Target Q-value for first next_observation: 1 * -5 + 1 * 6 + 1 =
2
Target Q-value for second next_observation: 1 * -7 + 1 * 8 + 1 =
2
TD targets: 10 + 0.9 * 2 = 11.8 and 20 + 0.9 * 2 = 21.8
TD errors: 11.8 - 5 = 6.8 and 21.8 - 8 = 13.8
TD loss: 6.3 and 13.3 (Huber loss subtracts 0.5)
Overall loss: (6.3 + 13.3) / 2 = 9.8
```

Refer to the following code listing:

```
class DqnAgentSampleOne(DqnAgentBase):
  def setUp(self):
    super(DqnAgentSampleOne, self).setUp()
  def runLossWithChangedOptimalActions(self):
    q_net = DummyNet(self._observation_spec, self._action_spec)
    agent = tf_agents.agents.dqn.dqn_agent.DdqnAgent(
      self._time_step_spec,
      self._action_spec,
      q_network=q_net,
      optimizer=None)
    observations = tf.constant([[1, 2], [3, 4]], dtype=tf.float32)
    time_steps = ts.restart(observations, batch_size=2)
    actions = tf.constant([0, 1], dtype=tf.int32)
    action_steps = policy_step.PolicyStep(actions)
    rewards = tf.constant([10, 20], dtype=tf.float32)
    discounts = tf.constant([0.9, 0.9], dtype=tf.float32)
    # Note that instead of [[5, 6], [7, 8]] as before, we now have
    -5 and -7.
    next_observations = tf.constant([[-5, 6], [-7, 8]], dtype=tf.
    float32)
    next_time_steps = ts.transition(next_observations, rewards,
    discounts)
```

```
experience = trajectories_test_utils.stacked_trajectory_from_
transition(
   time_steps, action_steps, next_time_steps)
loss, _ = agent._loss(experience)
self.evaluate(tf.compat.v1.global_variables_initializer())
loss_evaluate = self.evaluate(loss)
tf.print("loss:", loss, output_stream=sys.stdout)
```

DQN based agent for CartPole game

Let us use the **`tf-agent`** library with the CartPole game using OpenAI gym environment, to create a DQN based agent which plays the CartPole game:

```
! sudo apt-get install -y xvfb ffmpeg
! pip install -q 'imageio==2.4.0'
! pip install -q pyvirtualdisplay
! pip install -q tf-agents
```

- **Hyperparameters**: Set the hyperparameters:

```
num_iterations = 2000 # @param {type:"integer"}
initial_collect_steps = 100 # @param {type:"integer"}
collect_steps_per_iteration = 1 # @param {type:"integer"}
replay_buffer_max_length = 100000 # @param {type:"integer"}
batch_size = 64 # @param {type:"integer"}
learning_rate = 1e-3 # @param {type:"number"}
log_interval = 200 # @param {type:"integer"}
num_eval_episodes = 10 # @param {type:"integer"}
eval_interval = 1000 # @param {type:"integer"
```

- **Environment**: Load the CartPole environment from the OpenAI gym suite:

```
env_name = 'CartPole-v0'
env = suite_gym.load(env_name)
```

  Let us render this environment using PIL library to see how it looks. A free-swinging pole is attached to a cart. The goal is to move the cart right or left in order to keep the pole pointing up. Once the environment is loaded, you can see a figure similar to *figure 8.14*:

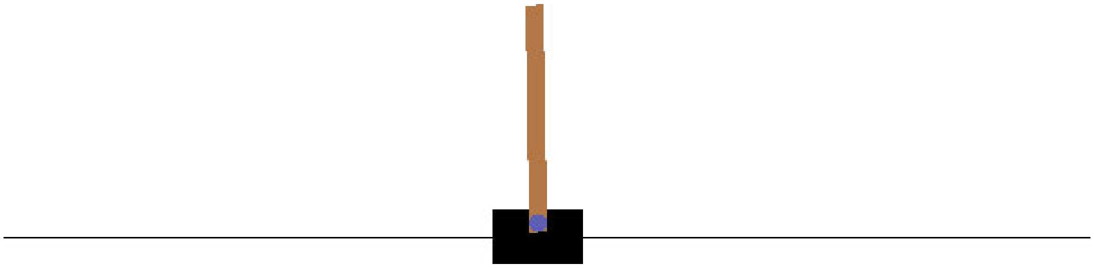


Figure 8.14: *CartPole environment loaded*

The **`action_spec()`** method returns the shape, data types, and allowed values of valid actions:

```
print('Action Spec:')
print(env.action_spec())
```

```
Action Spec:
BoundedArraySpec(shape=(), dtype=dtype('int64'), name='action',
minimum=0, maximum=1)
```

Let us look at the details of the CartPole environment:

- **`observation`** is an array of four floats:
  - the position and velocity of the cart
  - the angular position and velocity of the pole
- **`reward`** is a scalar float value.
- **`action`** is a scalar integer with only two possible values:
  - `0`: move left
  - `1`: move right

Two environments - one for **training** other for **testing**:

```
train_py_env = suite_gym.load(env_name)
eval_py_env = suite_gym.load(env_name)
```

Convert the environments from Python to TensorFlow:

```
train_env = tf_py_environment.TFPyEnvironment(train_py_env)
eval_env = tf_py_environment.TFPyEnvironment(eval_py_env)
```

Agent

An agent represents the algorithm used to solve an RL problem. TF-agents provide standard implementations for the following agents:

- DQN (**https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf**)
- REINFORCE (**https://www-anw.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf**)
- DDPG (**https://arxiv.org/pdf/1509.02971.pdf**)
- TD3 (**https://arxiv.org/pdf/1802.09477.pdf**)
- PPO (**https://arxiv.org/abs/1707.06347**)
- SAC (**https://arxiv.org/abs/1801.01290**)

The DQN agent can be used in an environment which has a discrete action space.

DQN agent contains a **QNetwork**, a neural network model that can learn to predict **QValues** (expected returns) for all actions, given an observation from the environment.

We will be using **`tf_agents.networks`** to create a QNetwork. The network consists of a sequence of **`tf.keras.layers.Dense`** layers, where the final layer will have one output for each possible action:

```
fc_layer_params = (100, 50)
action_tensor_spec = tensor_spec.from_spec(env.action_spec())
num_actions = action_tensor_spec.maximum - action_tensor_spec.minimum
+ 1
# Define a helper function to create Dense layers configured with the
right
# activation and kernel initializer.
def dense_layer(num_units):
  return tf.keras.layers.Dense(
    num_units,
    activation=tf.keras.activations.relu,
    kernel_initializer=tf.keras.initializers.VarianceScaling(
      scale=2.0, mode='fan_in', distribution='truncated_normal'))
# QNetwork consists of a sequence of Dense layers followed by a dense
layer
# with `num_actions` units to generate one q_value per available
action as
# its output.
dense_layers = [dense_layer(num_units) for num_units in
fc_layer_params]
q_values_layer = tf.keras.layers.Dense(
  num_actions,
  activation=None,
  kernel_initializer=tf.keras.initializers.RandomUniform(
    minval=-0.03, maxval=0.03),
  bias_initializer=tf.keras.initializers.Constant(-0.2))
q_net = sequential.Sequential(dense_layers + [q_values_layer])
```

Note that variance scaling helps network learn the weights better.

`DqnAgent` is initialized using **`tf_agents.agents.dqn.dqn_agent`**. It takes **`time_step_spec`**, **`action_spec`**, **`QNetwork`**, and **`optimizer`** (**`AdamOptimizer`**) in this case:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
train_step_counter = tf.Variable(0)
```

```
agent = dqn_agent.DqnAgent(
  train_env.time_step_spec(),
  train_env.action_spec(),
  q_network=q_net,
  optimizer=optimizer,
  td_errors_loss_fn=common.element_wise_squared_loss,
  train_step_counter=train_step_counter)
agent.initialize()
```

Next, let us look at the policy.

Policy

A policy is used to define the way an agent acts in an environment. The goal of reinforcement learning is to train the model so that the policy produces the desired outcome.

In this example:

- The desired outcome is keeping the pole balanced upright over the cart.
- The policy returns an action (left or right) for each **`time_step`** observation.

Agents contain two policies:

- **`agent.policy`**: the main policy that is used for evaluation and deployment
- **`agent.collect_policy`**: a second policy that is used for data collection:
  - `eval_policy = agent.policy`
  - `collect_policy = agent.collect_policy`

Policies can be created independent of the agent:

```
random_policy =
random_tf_policy.RandomTFPolicy(train_env.time_step_spec(),
train_env.action_spec())
example_environment = tf_py_environment.TFPyEnvironment(
  suite_gym.load('CartPole-v0'))
time_step = example_environment.reset()
random_policy.action(time_step)
```

Next, we will look at how to evaluate effectiveness of the policy.

Measures and metrics

Average return is used to evaluate policy. It is the sum of rewards obtained by running a policy in an environment for an episode. Several episodes lead to an average return.

The following function computes average return of policy for a given policy, environment, and number of episodes:

```
#@test {"skip": true}
def compute_avg_return(environment, policy, num_episodes=10):
total_return = 0.0
 for _ in range(num_episodes):
 time_step = environment.reset()
 episode_return = 0.0
 while not time_step.is_last():
  action_step = policy.action(time_step)
  time_step = environment.step(action_step.action)
  episode_return += time_step.reward
 total_return += episode_return
avg_return = total_return / num_episodes
return avg_return.numpy()[0]
```

Replay buffer

The **replay** buffer is used to keep track of data collected from the environment. This example uses **tf_agents.replay_buffers.tf_uniform_replay_buffer.TFUniformReplayBuffer**.

The constructor of the class requires the specs for the data it will be collecting. This is available from the agent using the **collect_data_spec** method. The batch size and maximum buffer length are also required:

```
replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
  data_spec=agent.collect_data_spec,
  batch_size=train_env.batch_size,
  max_length=replay_buffer_max_length)
```

Data collection

Accessing **replay_buffer** as a dataset:

```
dataset = replay_buffer.as_dataset(
num_parallel_calls=3,
  sample_batch_size=batch_size,
  num_steps=2).prefetch(3)
```

dataset has the following runtime attributes:

```
<PrefetchDataset shapes: (Trajectory(step_type=(64, 2), observation=
(64, 2, 4), action=(64, 2), policy_info=(), next_step_type=(64, 2),
reward=(64, 2), discount=(64, 2)), BufferInfo(ids=(64, 2),
probabilities=(64,))), types: (Trajectory(step_type=tf.int32,
observation=tf.float32, action=tf.int64, policy_info=(),
next_step_type=tf.int32, reward=tf.float32, discount=tf.float32),
BufferInfo(ids=tf.int64, probabilities=tf.float32))>
```

Training the agent

During the training loop, two things happen:

- collecting data from the environment
- using the data to train the agent's neural network(s)

The following code also periodically evaluates the policy and prints the current score. The code sample will take ~*5 minutes* to run:

```
try:
  %%time
except:
  pass
# (Optional) Optimize by wrapping some of the code in a graph using
TF function.
agent.train = common.function(agent.train)
# Reset the train step.
agent.train_step_counter.assign(0)
# Evaluate the agent's policy once before training.
avg_return = compute_avg_return(eval_env, agent.policy,
num_eval_episodes)
returns = [avg_return]
for _ in range(num_iterations):
  # Collect a few steps using collect_policy and save to the replay
  buffer.
  collect_data(train_env, agent.collect_policy, replay_buffer,
    collect_steps_per_iteration)
  # Sample a batch of data from the buffer and update the agent's
  network.
  experience, unused_info = next(iterator)
  train_loss = agent.train(experience).loss
  step = agent.train_step_counter.numpy()
  if step % log_interval == 0:
  print('step = {0}: loss = {1}'.format(step, train_loss))
  if step % eval_interval == 0:
    avg_return = compute_avg_return(eval_env, agent.policy,
    num_eval_episodes)
    print('step = {0}: Average Return = {1}'.format(step, avg_return))
    returns.append(avg_return)
```

The output shows the average return as a function of time:

```
step = 200: loss = 24.947084426879883
step = 400: loss = 15.076136589050293
step = 600: loss = 20.803768157958984
step = 800: loss = 153.6983184814453
step = 1000: loss = 38.69990921020508
step = 1000: Average Return = 153.10000610351562
step = 1200: loss = 36.78254699707031
…
step = 18000: loss = 75024480.0
step = 18000: Average Return = 178.39999389648438
step = 18200: loss = 95447344.0
step = 18400: loss = 102358096.0
step = 18600: loss = 283645856.0
step = 18800: loss = 107610192.0
step = 19000: loss = 57826464.0
step = 19000: Average Return = 193.60000610351562
step = 19200: loss = 166127808.0
step = 19400: loss = 140119344.0
step = 19600: loss = 315532032.0
step = 19800: loss = 155458912.0
step = 20000: loss = 138126432.0
step = 20000: Average Return = 187.6999969482422
```

Plotting the output:

```
#@test {"skip": true}
iterations = range(0, num_iterations + 1, eval_interval)
plt.plot(iterations, returns)
plt.ylabel('Average Return')
plt.xlabel('Iterations')
plt.ylim(top=250)
```

Output will be similar to *figure 8.15* for average return as a function of iterations:

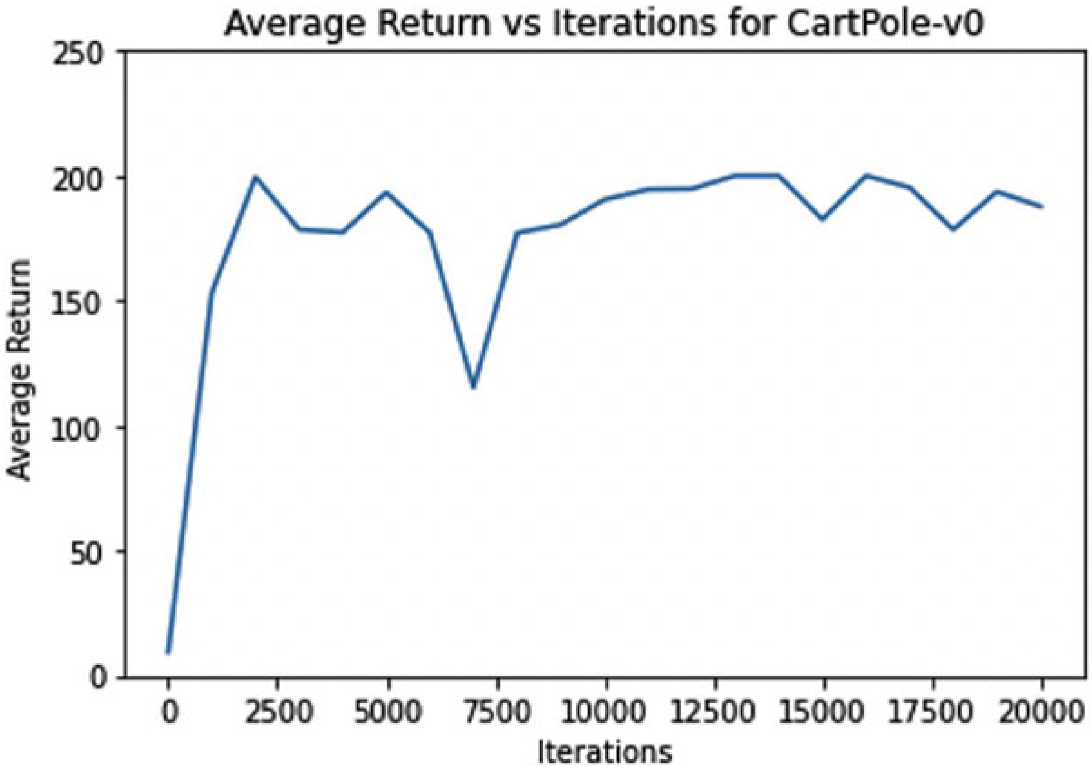


Figure 8.15: *Average return versus iterations for CartPole-v0 with DQN network using TF-agents*

Removing variance scaling from the model

Let us remove the variation scaling from the dense layer and see the output:

```
fc_layer_params = (100, 50)
action_tensor_spec = tensor_spec.from_spec(env.action_spec())
num_actions = action_tensor_spec.maximum - action_tensor_spec.minimum
+ 1
# Define a helper function to create Dense layers configured with the
right
# activation and kernel initializer.
def dense_layer(num_units):
  tf.keras.layers.Dense(
    num_units,
    activation=tf.keras.activations.relu
    )
# QNetwork consists of a sequence of Dense layers followed by a dense
layer
# with `num_actions` units to generate one q_value per available
action as
# Its output.
```

```
dense_layers = [dense_layer(num_units) for num_units in
fc_layer_params]
q_values_layer = tf.keras.layers.Dense(
  num_actions,
  activation=None,
  kernel_initializer=tf.keras.initializers.RandomUniform(
    minval=-0.03, maxval=0.03),
  bias_initializer=tf.keras.initializers.Constant(-0.2))
q_net = sequential.Sequential(dense_layers + [q_values_layer])
```

The following plot shows that not having variance scaling leads to average return reaching **200** very soon:

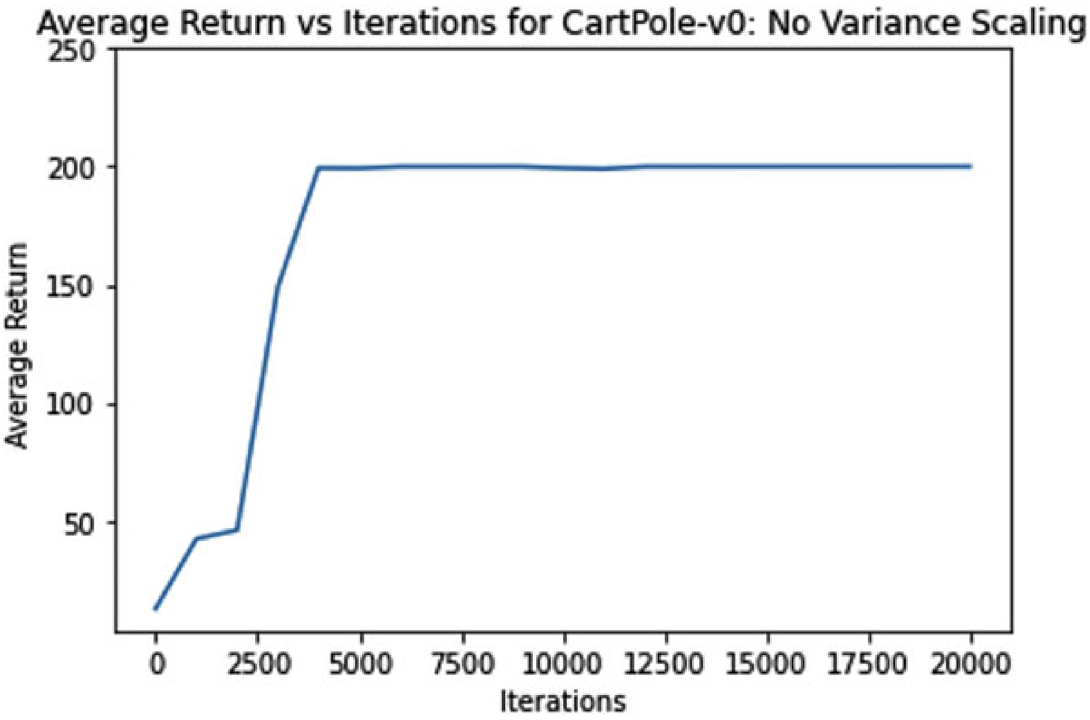


Figure 8.16: *Average return versus iterations for CartPole-v0 with no variance scaling, using DQN network, and TF-agents*

In the next section, we will look at the SAC agents supported in the TF-agents' library.

SAC agent's support in TF-agents

In this section, we will look at how to create a basic actor critic agent, and then how to use a trajectory to train it. This is the same agent type that we built earlier without using TF-Agent library.

TF-agent provides out-of-box support, as we learnt in the earlier sections:

```
class MyActorPolicy(object):
      def __init__(self,
        time_step_spec,
        action_spec,
        actor_network,
        training=False):
   del time_step_spec
   del actor_network
   del training
   single_action_spec = tf.nest.flatten(action_spec)[0]
   # Action is maximum of action range.
   self._action = single_action_spec.maximum
   self._action_spec = action_spec
   self.info_spec = ()
      def action(self, time_step):
   observation = time_step.observation
   batch_size = observation.shape[0]
   action = tf.constant(self._action, dtype=tf.float32,
     shape=[batch_size, 1])
   return policy_step.PolicyStep(action=action)
      def distribution(self, time_step, policy_state=()):
   del policy_state
   action = self.action(time_step).action
   return policy_step.PolicyStep(action=_MockDistribution(action))
      def get_initial_state(self, batch_size):
   del batch_size
   return ()
```

Create a critic network and implement **`__init__()`**, **`copy()`**, and **`call()`** methods. In **`__init__()`**, we initiate a **`input_tensor_spec`**, **`state_spec`**, and **`_l2_regularization_weight`**.

We implement the following three layers:

- **`_value_layer`**
- **`_shared_layer`**
- **`_action_layer`**

```
self._value_layer = tf.keras.layers.Dense(
     1,
     kernel_regularizer=tf.keras.regularizers.l2(l2_regularization_wei
     ght),
     kernel_initializer=tf.constant_initializer([[0], [1]]),
     bias_initializer=tf.constant_initializer([[0]]))
self._shared_layer = shared_layer
self._action_layer = tf.keras.layers.Dense(
    1,
     kernel_regularizer=
     tf.keras.regularizers.l2(l2_regularization_weight),
     kernel_initializer=tf.constant_initializer([[1]]),
     bias_initializer=tf.constant_initializer([[0]]))
```

The call function

Calculate the **s_value (state)**, **a_value (actions)**, and **q_value** based on the previous actions and values passed as inputs:

```
  def call(self, inputs, step_type, network_state=()):
  del step_type
    observation, actions = inputs
    actions = tf.cast(tf.nest.flatten(actions)[0], tf.float32)
    states = tf.cast(tf.nest.flatten(observation)[0], tf.float32)
    s_value = self._value_layer(states)
  if self._shared_layer:
      s_value = self._shared_layer(s_value)
    a_value = self._action_layer(actions)
  # Biggest state is best state.
  q_value = tf.reshape(s_value + a_value, [-1])
    return q_value, network_state
```

The complete code listing is as follows:

```
class DummyCriticNet(network.Network):
  def __init__(self, l2_regularization_weight=0.0, shared_layer=None):
  super(DummyCriticNet, self).__init__(
    input_tensor_spec=(tensor_spec.TensorSpec([2], tf.float32),
          tensor_spec.TensorSpec([1], tf.float32)),
      state_spec=(),
    name=None)
  self._l2_regularization_weight = l2_regularization_weight
  self._value_layer = tf.keras.layers.Dense(
    1,
kernel_regularizer=
tf.keras.regularizers.l2(l2_regularization_weight),
    kernel_initializer=tf.constant_initializer([[0], [1]]),
    bias_initializer=tf.constant_initializer([[0]]))
  self._shared_layer = shared_layer
self._action_layer = tf.keras.layers.Dense(
 1,
   kernel_regularizer=
      tf.keras.regularizers.l2(l2_regularization_weight),
 kernel_initializer=tf.constant_initializer([[1]]),
 bias_initializer=tf.constant_initializer([[0]]))
def copy(self, name=''):
del name
return DummyCriticNet(
 l2_regularization_weight=self._l2_regularization_weight,
 shared_layer=self._shared_layer)
def call(self, inputs, step_type, network_state=()):
del step_type
 observation, actions = inputs
actions = tf.cast(tf.nest.flatten(actions)[0], tf.float32)
states = tf.cast(tf.nest.flatten(observation)[0], tf.float32)
s_value = self._value_layer(states)
if self._shared_layer:
 s_value = self._shared_layer(s_value)
a_value = self._action_layer(actions)
# Biggest state is best state.
q_value = tf.reshape(s_value + a_value, [-1])
return q_value, network_state
```

Create SAC agent sample

In the following sample, we create a simple SAC agent with `_time_step_spec`, `_action_spec`, `critic_network`, and `ActorPolicy` being passed in the constructor:

- `_time_step_spec` is based on `_obs_spec`, which varies between `0` and `1`
- `._action_spec` varies between `-1` and `1`

All the optimizers are of the type `tf.compat.v1.train.AdamOptimizer(0.001)`:

```
 class SacAgentSample( test_utils.TestCase):
  def setUp(self):
  super(SacAgentSample, self).setUp()
  self._obs_spec = tensor_spec.BoundedTensorSpec([2],
          tf.float32,
          minimum=0,
          maximum=1)
  self._time_step_spec = ts.time_step_spec(self._obs_spec)
  self._action_spec = tensor_spec.BoundedTensorSpec([1], tf.float32,
  -1,
 1)
  def runCreateAgent(self, create_critic_net_fn, skip_in_tf1):
   critic_network = create_critic_net_fn()
   SacAgent(
   self._time_step_spec,
   self._action_spec,
    critic_network=critic_network,
   actor_network=None,
  actor_optimizer=
 tf.compat.v1.train.AdamOptimizer(0.001),
     critic_optimizer=
 tf.compat.v1.train.AdamOptimizer(0.001),
     alpha_optimizer=
 tf.compat.v1.train.AdamOptimizer(0.001))
    actor_policy_ctor=MyActorPolicy)
```

Create an agent and train on trajectory

Create a **`trajectory_spec`** from **`trajectory.Trajectory`**. This is used to create a **`sample_trajectory`** which is passed to the **`agent.train()`**:

```
def runAgentTrajectoryTrain(self):
  actor_net = actor_distribution_network.ActorDistributionNetwork(
  self._obs_spec,
  self._action_spec,
  fc_layer_params=(10,),
   continuous_projection_net=tanh_normal_projection_network
   .TanhNormalProjectionNetwork)
  agent = SacAgent(
  self._time_step_spec,
  self._action_spec,
   critic_network=MyCriticNet(),
   actor_network=actor_net,
  actor_optimizer=tf.compat.v1.train.AdamOptimizer(0.001),
  critic_optimizer=tf.compat.v1.train.AdamOptimizer(0.001),
  alpha_optimizer=tf.compat.v1.train.AdamOptimizer(0.001))
  trajectory_spec = trajectory.Trajectory(
  step_type=self._time_step_spec.step_type,
  observation=self._time_step_spec.observation,
  action=self._action_spec,
   policy_info=(),
  next_step_type=self._time_step_spec.step_type,
   reward=tensor_spec.BoundedTensorSpec(
   [], tf.float32, minimum=0.0, maximum=1.0, name='reward'),
  discount=self._time_step_spec.discount)
  sample_trajectory_experience = tensor_spec.sample_spec_nest(
  trajectory_spec, outer_dims=(3, 2))
  loss_info = agent.train(sample_trajectory_experience)
tf.print(loss_info)
```

Output of the preceding **`tf.print`** will give the loss, **`critic_loss`**, **`actor_loss`**, and **`alpha_loss`**:

```
LossInfo(loss=3.35792017, extra=SacLossInfo(critic_loss=2.24238253,
actor_loss=1.11553776, alpha_loss=0))
```

Reinforce agent

In this section, we look at how to use reinforce agent using **`tf-agents`** framework. Most of the steps are same as the preceding example for *CartPole-v0*:

1. We define an **`ActorDistributionNetwork`**:

   ```
   actor_net = actor_distribution_network.ActorDistributionNetwork(
     train_env.observation_spec(),
     train_env.action_spec(),
     fc_layer_params=fc_layer_params)
   ```

2. Define a reinforce agent:

   ```
   optimizer =
   tf.compat.v1.train.AdamOptimizer(learning_rate=learning_rate)
   train_step_counter = tf.compat.v2.Variable(0)
   tf_agent = reinforce_agent.ReinforceAgent(
     train_env.time_step_spec(),
     train_env.action_spec(),
     actor_network=actor_net,
     optimizer=optimizer,
     normalize_returns=True,
     train_step_counter=train_step_counter)
   tf_agent.initialize()
   Training the agent
   Let us look at the step where we train the target:
   #@test {"skip": true}
   try:
   %%time
   ```

```
except:
pass
# (Optional) Optimize by wrapping some of the code in a graph
using TF function.
tf_agent.train = common.function(tf_agent.train)
# Reset the train step
tf_agent.train_step_counter.assign(0)
# Evaluate the agent's policy once before training.
avg_return = compute_avg_return(eval_env, tf_agent.policy,
num_eval_episodes)
returns = [avg_return]
for _ in range(num_iterations):
 # Collect a few episodes using collect_policy and save to the
 replay buffer.
 collect_episode(
   train_env, tf_agent.collect_policy, collect_episodes_per_
   iteration)
 # Use data from the buffer and update the agent's network.
 experience = replay_buffer.gather_all()
 train_loss = tf_agent.train(experience)
 replay_buffer.clear()
 step = tf_agent.train_step_counter.numpy()
 if step % log_interval == 0:
 print('step = {0}: loss = {1}'.format(step, train_loss.loss))
 if step % eval_interval == 0:
 avg_return = compute_avg_return(eval_env, tf_agent.policy,
 num_eval_episodes)
 print('step = {0}: Average Return = {1}'.format(step, avg_
 return))
 returns.append(avg_return)
```

Output of the preceding training step:

```
CPU times: user 0 ns, sys: 2 µs, total: 2 µs
Wall time: 4.77 µs
step = 25: loss = 0.006921291351318359
step = 50: loss = 0.03737598657608032
step = 50: Average Return = 84.9000015258789
step = 75: loss = 0.11638593673706055
step = 100: loss = -0.6089606285095215
step = 100: Average Return = 130.3000030517578
step = 125: loss = 0.6521391868591309
step = 150: loss = -1.0057716369628906
step = 150: Average Return = 198.8000030517578
step = 175: loss = -3.2779808044433594
step = 200: loss = -3.663456916809082
step = 200: Average Return = 200.0
step = 225: loss = -2.1488280296325684
step = 250: loss = 1.3752391338348389
step = 250: Average Return = 197.89999389648438
```

Plotting the output

Let us plot the preceding output generated and stored in returns:

```
steps = range(0, num_iterations + 1, eval_interval)
plt.plot(steps, returns)
plt.ylabel('Average Return')
plt.xlabel('Step')
plt.ylim(top=250)
```

The average return reaches **200** in about *130 steps,* as shown in the *figure 8.17*:

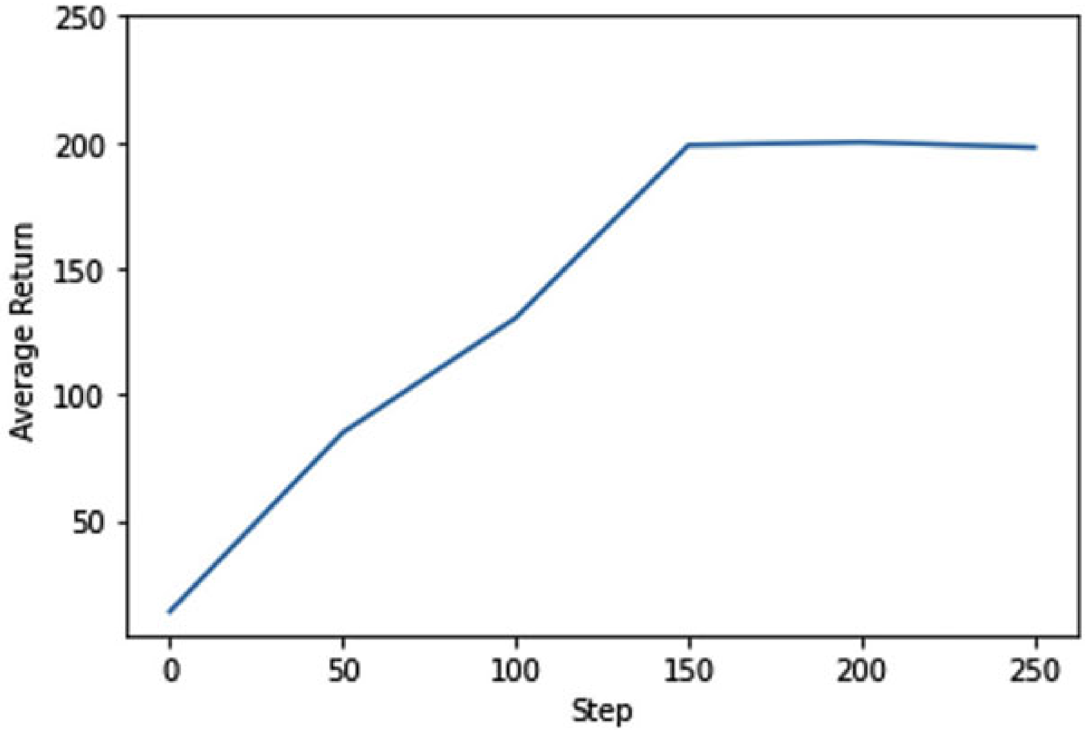


Figure 8.17: *Average return for SAC agent as a function of steps*

Conclusion

In this chapter, we learnt the basics of reinforcement learning. We started with defining the policy value. This was followed by concepts like policy evaluation and policy iterations. We defined the function Q and its approximation using neural network – **DQN**. This was followed by more advanced concepts like double DQN. We looked at actor-critic network which is a combination of policy and value-based learning. We looked at TF-agent library, which makes it easier to write RL algorithms using TensorFlow.

Questions

1. **Reinforcement learning is:**
    a. unsupervised learning
    b. supervised learning
    c. award-based learning
    d. none of the above
2. **Which of the following is true about reinforcement learning?**
    a. The agent gets rewards or penalty according to the action.
    b. It's online learning.
    c. The target of an agent is to maximize the rewards.
    d. all of the above
3. **Hidden Markov Model is used in:**
    a. supervised learning
    b. unsupervised learning
    c. reinforcement learning
    d. all of the above

Code listing

Base directory is as follows:

https://github.com/rajdeepd/tensorflow_2.0_book_code/tree/master/ch09

References

- **Understanding Stabilizing Experience Replay for Deep Multi-Agent Reinforcement Learning**: https://medium.com/@parnianbrk/understanding-stabilising-experience-replay-for-deep-multi-agent-reinforcement-learning-84b4c04886b5.
- **REINFORCE algorithm from (Williams, 1992)**: https://www-anw.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf.
- **Soft Actor-Critic Algorithms and Applications**: https://arxiv.org/abs/1812.05905.
- **Addressing Function Approximation Error in Actor-Critic Methods by Fujimoto et al**: https://arxiv.org/abs/1802.09477.
- **DQN C51/Rainbow**: https://www.tensorflow.org/agents/tutorials/9_c51_tutorial.
- **Human-level control through deep reinforcement learning**: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf.

- **Lecture 1: Introduction to RL**: **https://web.stanford.edu/class/cs234/slides/lecture1.pdf**.
- **Stanford CS234: Reinforcement Learning | Winter 2019 | Lecture 1 – Introduction**: **https://www.youtube.com/watch?v=FgzM3zpZ55o&list=PLoROMvodv4rOSOPzutgyCTapiGlY2Nd8u**.
- **Epsilon-Greedy Algorithm in Reinforcement Learning**: **https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/**.
- **Automate your workflow from idea to production**: **https://github.com/marload/DeepRL-TensorFlow2**.
- **David-Silver-Reinforcement-learning**: **https://github.com/dalmia/David-Silver-Reinforcement-learning**.

Appendix

Policy evaluation

Policy evaluation to achieve optimal policy is based on the following steps:

Iterative application of Bellman expectation backup:

- v_1 ---> v_2 ---> ... $v\pi$
- Using synchronous backups:
  - At each iteration k + 1, for all states s ∈ S
  - Update $v_{k+1}(s)$ from $v_k(s')$ where s' is a successor state of s

1. Let us look at the implementation in Python:

```
from IPython.core.debugger import set_trace
import numpy as np
import pprint
import sys
if "../" not in sys.path:
  sys.path.append("../")
from lib.envs.gridworld import GridworldEnv
```

2. Instantiate the environment:

```
pp = pprint.PrettyPrinter(indent=2)
env = GridworldEnv()
```

3. Define policy **`eval`** function:

```
def policy_eval(policy, env, discount_factor=1.0,
theta=0.00001):
 # Start with a random (all 0) value function
 V = np.zeros(env.nS)
 while True:
 delta = 0
 # For each state, perform a "full backup"
 for s in range(env.nS): v = 0
 # Look at the possible next actions
 for a, action_prob in enumerate(policy[s]):
  # For each action, look at the possible next states...
  for prob, next_state, reward, done in env.P[s][a]:
  # Calculate the expected value. Ref: Sutton book eq. 4.6.
  v += action_prob * prob * (reward + discount_factor \
    * V[next_state])
  # How much our value function changed (across any states)
  delta = max(delta, np.abs(v - V[s])) V[s] = v
  # Stop evaluating once our value function change is below
  # a threshold
  if delta < theta:
   break return np.array(V)
```

4. Create a random policy, and find the value function:

```
random_policy = np.ones([env.nS, env.nA]) / env.nA
v = policy_eval(random_policy, env)
```

5. Let us print the values we get for **v**:

```
Value Function:
[ 0. -13.99993529 -19.99990698 -21.99989761 -13.99993529
 -17.9999206 -19.99991379 -19.99991477 -19.99990698 -19.99991379
 -17.99992725 -13.99994569 -21.99989761 -19.99991477
 -13.99994569
 0. ]
```

6. Reshaped grid value function:

```
[[ 0. -13.99993529 -19.99990698 -21.99989761]
 [-13.99993529 -17.9999206 -19.99991379 -19.99991477]
 [-19.99990698 -19.99991379 -17.99992725 -13.99994569]
 [-21.99989761 -19.99991477 -13.99994569 0. ]]
```

Policy iteration

Evaluate a policy given an environment and a full description of the environment's dynamics:

```
def policy_eval(policy, env, discount_factor=1.0, theta=0.00001):
  # Start with a random (all 0) value function
  V = np.zeros(env.nS)
  while True:
    delta = 0
    # For each state, perform a "full backup"
    for s in range(env.nS):
      v = 0
      # Look at the possible next actions
      for a, action_prob in enumerate(policy[s]):
          # For each action, look at the possible next states...
          for prob, next_state, reward, done in env.P[s][a]:
            # Calculate the expected value
              v += action_prob * prob * \
              (reward + discount_factor * V[next_state])
        # How much our value function changed (across any states)
      delta = max(delta, np.abs(v - V[s]))
      V[s] = v
    # Stop evaluating once our value function change is below a
    # threshold
    if delta < theta:
        break
  return np.array(V)
```

Policy improvement algorithm iteratively evaluates and improves a policy until an optimal policy is found:

```
def one_step_lookahead(env, discount_factor,state, V):
      """
      Helper function to calculate the value for all action
      in a given state.
      Args:
        state: The state to consider (int)
        V: The value to use as an estimator, Vector of
          length env.nS
      Returns:
        A vector of length env.nA containing the expected value
        of each action.
      """
      A = np.zeros(env.nA)
      for a in range(env.nA):
```

```
      for prob, next_state, reward, done in env.P[state][a]:
        A[a] +=
          prob * (reward + discount_factor * V[next_state])
    return A
def policy_improvement(env, policy_eval_fn=policy_eval,
discount_factor=1.0):
  """
"""
  # Start with a random policy
  policy = np.ones([env.nS, env.nA]) / env.nA
  V = None
  while True:
    # V = policy_eval_fn(policy, env, discount_factor=1.0,
    # theta=0.00001)
    # Implement this!
    # For each state, perform a "full backup"
    V = policy_eval_fn(policy, env, discount_factor)
    # Will be set to false if we make any changes to the policy
    policy_stable = True
    for s in range(env.nS):
      chosen_a = np.argmax(policy[s])
      action_values = one_step_lookahead(env,
        discount_factor,s, V)
      best_a = np.argmax(action_values)
      if chosen_a != best_a:
        policy_stable = False
      policy[s] = np.eye(env.nA)[best_a]
    if policy_stable:
    return policy, V
policy, v = policy_improvement(env)
print("Policy Probability Distribution:")
print(policy)
print("")
```

Output of the probability distribution:

```
Policy Probability Distribution: [[1. 0. 0. 0.] [0. 0. 0. 1.] [0. 0.
0. 1.] [0. 0. 1. 0.] [1. 0. 0. 0.] [1. 0. 0. 0.] [1. 0. 0. 0.] [0. 0.
1. 0.] [1. 0. 0. 0.] [1. 0. 0. 0.] [0. 1. 0. 0.] [0. 0. 1. 0.] [1. 0.
0. 0.] [0. 1. 0. 0.] [0. 1. 0. 0.] [1. 0. 0. 0.]]
```

Reshape the grid policy and value function:

```
print("Reshaped Grid Policy (0=up, 1=right, 2=down, 3=left):")
print(np.reshape(np.argmax(policy, axis=1), env.shape))
print("")
print("Value Function:")
print(v)
print("")
print("Reshaped Grid Value Function:")
print(v.reshape(env.shape))
print("")
```

```
Reshaped Grid Policy (0=up, 1=right, 2=down, 3=left):
[[0 3 3 2]
[0 0 0 2]
[0 0 1 2]
[0 1 1 0]]
Value Function:
[ 0. -1. -2. -3. -1. -2. -3. -2. -2. -3. -2. -1. -3. -2. -1. 0.]
Reshaped Grid Value Function:
[[ 0. -1. -2. -3.]
  [-1. -2. -3. -2.]
  [-2. -3. -2. -1.]
  [-3. -2. -1. 0.]]
```

CHAPTER 9
Techniques to do Model Optimization

Introduction

The TensorFlow model optimization toolkit provides a set of tools for users, both beginners and advanced. It can be used to optimize machine learning models for deployment as well as execution.

It includes techniques like quantization and pruning for sparse weights. There are APIs built specifically for Keras based APIs.

For an overview of this project and individual tools, the optimization gains, and roadmap you can refer to the link:

https://www.tensorflow.org/model_optimization

Structure

In this chapter, we will cover the following topics:

- Background
- Extension to neural networks
- Quantization
- Post-training quantization
- Weight pruning

Objective

After going through this chapter, you will be able to understand how models can be compressed to make them more efficient without affecting the performance drastically.

Background

Neural networks are considered function approximators. They are trained to learn functions that capture underlying representations formulating the input data points/dataset. The neural network's weights and biases are referred to as *models' learnable parameters*. The weights are referred to as *coefficients of the function being learned*.

Let us consider the following function:

$$f(x) = x + 10x^2$$

In the preceding function, we have two terms on the right-hand side: *x* and x^2. The coefficients are *1* and *10* respectively. In the following plot, we can see that the behavior of the function does not change much when the first coefficient is nudged:

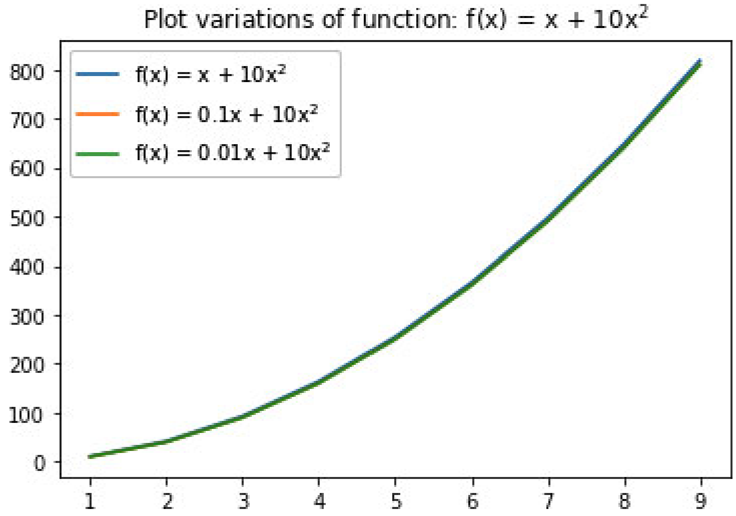


Figure 9.1: *Plot variations on changing multiplier for x in the quadratic equation*

This plot can be found at the following location: https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch09/pruning_example.ipynb.

Extension to neural networks

The preceding concept can be applied to neural networks as well. Consider the weights of a trained network. *How can we make sense of the weights that are non-significant?*

Let us look at the optimization process with a gradient descent. Weights are updated using different gradient magnitudes. The gradients of a given loss function are taken with respect to the weights and biases. Some of the weights get updated with larger gradient magnitudes (both positive and negative) than the others during optimization. These weights are considered significant by the optimizer to minimize the training objective. The weights that receive relatively smaller gradients are considered as non-significant.

After the training is complete, the weight magnitudes of a network are inspected layer by layer to find the weights that are significant. This decision can be made using several heuristics as listed below:

- Weight magnitudes are sorted in a descending manner to pick up the significant ones. This is combined with a sparsity level (percentage of weights to be pruned) one would want to achieve.
- We can also specify a threshold, and all the weights where magnitudes lie above that threshold will be considered as significant. This scheme can have several flavors:
    - The threshold can be a value that is the lowest for the entire network.
    - The threshold can be a value local to each of the layers inside a network. The significant weights are filtered out for each layer separately in this case.

Tools available

TensorFlow has been recently shipped with the model optimization tool: **TensorFlow-Model-Optimization**.

Advantages of model optimization

Model optimization helps to improve throughout and reduce the size of the model, which is especially useful in edge computing use cases.

As an example, if we take two *32-bit floating-point* matrices *A* and *B* and do a matrix multiplication, the resulting matrix is also a *32-bit floating-point* matrix:

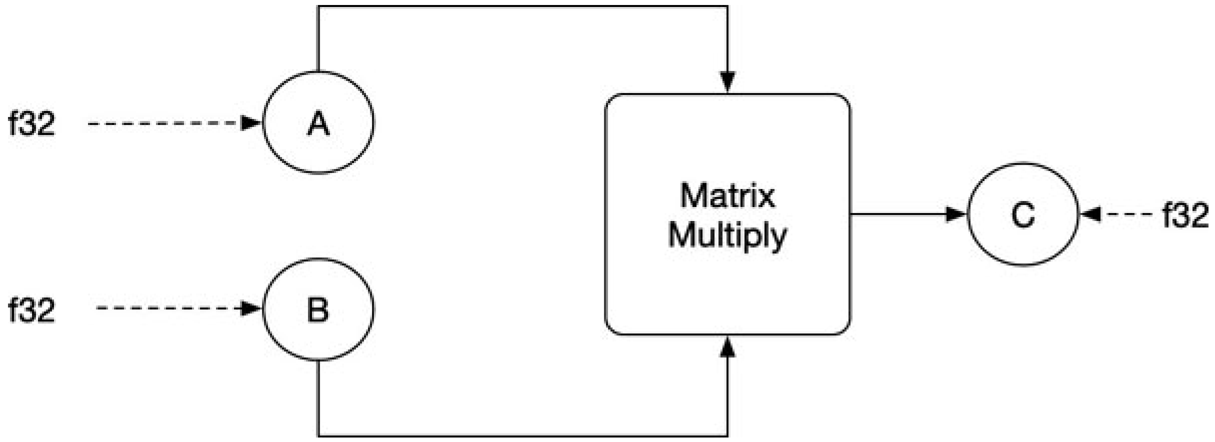


Figure 9.2: *Data types for two matrices before and after multiplication*

Operands are **f32** and the product is also **f32,** as shown in *figure 9.2* and *figure 9.3*:

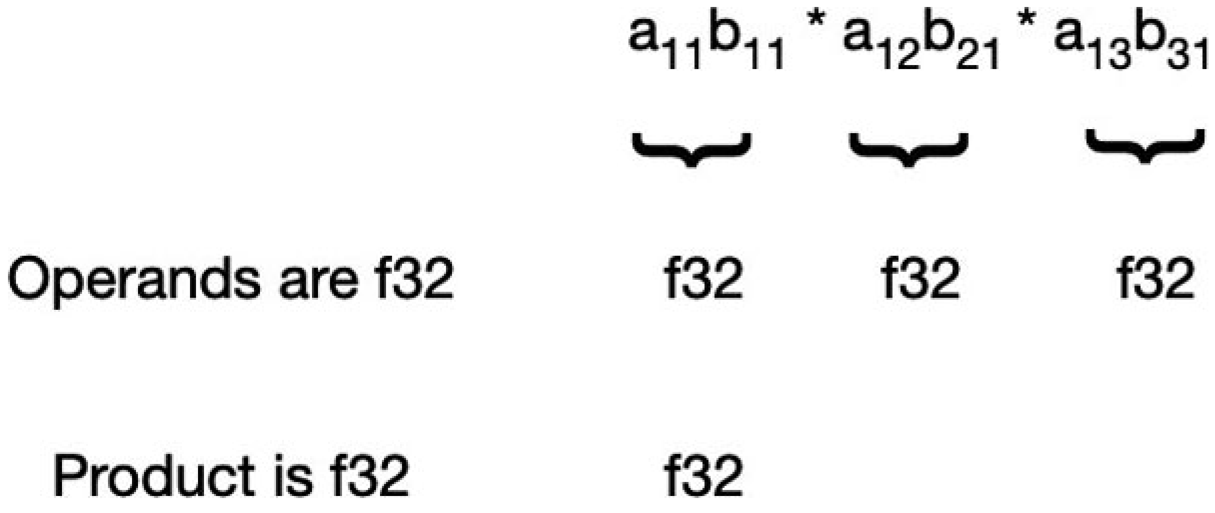


Figure 9.3: *Data types for two matrices before and after multiplication of each element a and b*

We can make the model more efficient by either reducing the precision of float from *32* to *8* or converting float *32* to integer. Advantages of lower precision from **f32** to **f8** are as follows:

- 4x smaller model
- faster integer operations
- less power consumed
- lower bandwidth

Quantization

Let us first define **quantization**. It is a process of model approximated representation using operators of lower precision.

Quantization is made up of techniques for performing computations and storing tensors at lower bit widths—floating point *8 bit* or integer than floating point precision of *32 bits*. A quantized model executes some or all the operations on tensors with integers instead of floating-point values. This technique allows for a more compact model and the use of high performance vectorized operations on many hardware platforms.

Deep learning frameworks, such as TensorFlow, have supported quantization natively using the toolkit described previously. The developers and data scientists have been using the built-in quantization modules successfully without knowing the inner workings of the technique. In this chapter, we will describe the mathematics of quantization for neural networks so that the readers would have some ideas about the quantization mechanisms.

Simple sample on quantization on *inception V3 network* using the TensorFlow:

```
import numpy as np
import tensorflow as tf
import inspect
tf.__version__
```

- Create a utility method **`_get_model()`** to create an **`InceptionV3`** model with input shape of **`(75,75,3)`**:

  ```
  model_type = 'InceptionV3'
  _MODEL_INPUT_SHAPES = {
      'InceptionV3': (75, 75, 3)
  }
  def _get_model(model_type):
    model_fn = [
     y for x, y in inspect.getmembers(tf.keras.applications)
     if x == model_type
    ][0]
    input_shape = _MODEL_INPUT_SHAPES.get(
     model_type, (32, 32, 3))
    return model_fn(weights=None, input_shape=input_shape)
  model = _get_model(model_type)
  ```

```
def _batch(dims, batch_size):
  if dims[0] is None:
    dims[0] = batch_size
  return dims
x_train = np.random.randn(
    *_batch(model.input.get_shape().as_list(),
    2)).astype('float32')
y_train = tf.keras.utils.to_categorical(
    np.random.randint(1000, size=(2, 1)), 1000)
```

- Create a quantized-aware model from the normal model using **`quantize_model(model)`** `method of` **`tensorflow_model_optimization.quantization.keras.quantize_model`**:

```
from tensorflow.python.keras import keras_parameterized
import tensorflow_model_optimization as tfmot
quantize_model = tfmot.quantization.keras.quantize_model
# q_aware stands for quantization aware.
q_aware_model = quantize_model(model)
```

In the next section, we explore how quantization can be done after training as well.

Post-training quantization

Post-training quantization is a set of general techniques to reduce CPU and hardware accelerator latency, processing, power, and model size with minimal degradation in model accuracy. These set of techniques can be performed on an already-trained float TensorFlow model, and applied during TensorFlow lite conversion as well.

Overview

TensorFlow Lite converts weights to *8-bit floating point* as it converts TensorFlow **graphdefs** (**https://haosdent.gitbooks.io/tensorflow-document/content/resources/data_versions.html**) to TensorFlow Lite flat buffer format. Dynamic range quantization (also called **post-training quantization**) is able to achieve *4x* reduction in model size. **TFLite** also supports using quantized kernels for faster implementation and mixing floating point kernels with quantized kernels in the same graph. Activations stored in floating point are quantized to *8-bits* of precision, and then are dequantized to floating point precision after processing.

In this technique, the weights are quantized post training. Model weights are not retrained to compensate for quantization; so make sure you test the model for accuracy.

Build a Fashion MNIST model

We will be creating a Fashion MNIST classification model and persist it.

Setup

1. Let us start by importing the relevant packages:

```
import logging
logging.getLogger("tensorflow").setLevel(logging.DEBUG)
import tensorflow as tf
from tensorflow import keras
import numpy as np
import pathlib
```

2. Train a TensorFlow model:

```
# Load Fashion MNIST dataset
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
 fashion_mnist.load_data()
# Normalize the input image so that each pixel value is
between 0 to 1.
train_images = train_images / 255.0
test_images = test_images / 255.0
# Define the model architecture
model = keras.Sequential([
 keras.layers.InputLayer(input_shape=(28, 28)),
 keras.layers.Reshape(target_shape=(28, 28, 1)),
 keras.layers.Conv2D(filters=12, kernel_size=(3, 3),
  activation=tf.nn.relu),
 keras.layers.MaxPooling2D(pool_size=(2, 2)),
 keras.layers.Flatten(),
 keras.layers.Dense(10)
])
# Train the digit classification model
model.compile(optimizer='ada
tfmot.sparsity.keras.prune_low_magnitude(m',
      loss=keras.losses.SparseCategoricalCrossentropy(
 from_logits=True),
      metrics=['accuracy'])
model.fit(
 train_images,
 train_labels,
 epochs=1,
 validation_data=(test_images, test_labels)
)
Downloading data from
https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
1875/1875 [==============================] - 23s 12ms/step
- loss: 0.5152 - accuracy: 0.8193 - val_loss: 0.4285 -
val_accuracy: 0.8513
```

For example, since you trained the model for just a single epoch, it only trains to ~*81%* to *83%* accuracy.

Convert to a TensorFlow Lite model

Using the Python **TFLiteConverter** (**https://www.tensorflow.org/lite/convert/python_api**), you can now convert the trained model into a TensorFlow Lite model:

1. Now, load the model using the `TFLiteConverter`:

   ```
   converter =
   tf.lite.TFLiteConverter.from_keras_model(model)
   tflite_model = converter.convert()
   INFO:tensorflow:Assets written to: /tmp/tmptdmqwtpq/assets
   ```

2. Write it out to a `tflite` file:

   ```
   tflite_models_dir =
   pathlib.Path("/tmp/fashion_mnist_tflite_models/")
   ```

```
tflite_models_dir.mkdir(exist_ok=True, parents=True)
tflite_model_file =
tflite_models_dir/"fashion_mnist_model.tflite"
tflite_model_file.write_bytes(tflite_model)
```

3. To quantize the model on export, set the optimizations flag to optimize for size.

   `tf.lite.Optimize.DEFAULT`: Default optimization strategy that quantizes model weights. Enhanced optimizations are gained by providing a representative dataset that quantizes biases and activations as well. Converter will do its best to reduce size and latency, while minimizing the loss in accuracy.

   ```
   converter.optimizations = [tf.lite.Optimize.DEFAULT]
   tflite_quant_model = converter.convert()
   tflite_model_quant_file =
    tflite_models_dir/"fashion_mnist_model.tflite"
   tflite_model_quant_file.write_bytes(tflite_quant_model)
   INFO:tensorflow:Assets written to: /tmp/tmpzqj403v1/assets
   23904
   ```

4. Let us look at the size of the model:

   ```
   !ls -lh {tflite_models_dir}
   total 24K
   -rw-r--r-- 1 root root 24K Sep 19 04:10
   fashion_mnist_model.tflite
   ```

   Notice how the resulting file is approximately *1/4* the size.

Run the TFLite models

Run the TensorFlow Lite model using the Python **TensorFlow Lite Interpreter**.

1. Load the model into an interpreter:

   ```
   interpreter =
   tf.lite.Interpreter(model_path=str(tflite_model_file))
   interpreter.allocate_tensors()
   interpreter_quant = tf.lite.Interpreter(model_path=str(
    tflite_model_quant_file))
   ```

```
interpreter_quant.allocate_tensors()
```

2. Test the model on one image:

```
test_image = np.expand_dims(test_images[1],
axis=0).astype(np.float32)
input_index = interpreter.get_input_details()[0]["index"]
output_index = interpreter.get_output_details()[0]
["index"]
interpreter.set_tensor(input_index, test_image)
interpreter.invoke()
predictions = interpreter.get_tensor(output_index)
```

3. Plot the true and predicted label:

```
import matplotlib.pylab as plt
plt.imshow(test_images[1])
template = "True:{true}, predicted:{predict}"
_ = plt.title(template.format(true= str(test_labels[1]),
predict=str(np.argmax(predictions[0]))))
plt.grid(False)
```

Plot in *figure 9. 4* shows the output of the preceding code as a grid with single image from **`test_images`**:

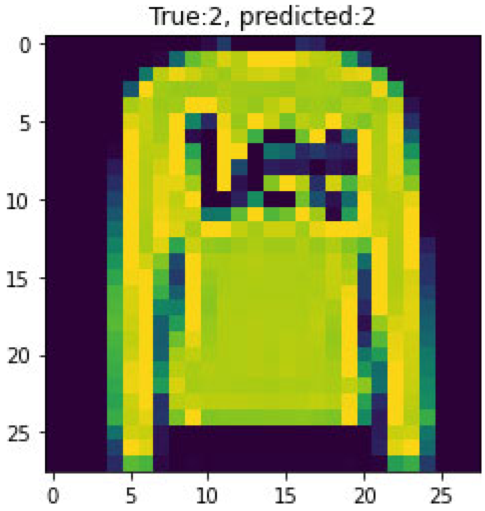


Figure 9.4: *Output of test_images[1] on executing the code*

Evaluate the models

We will evaluate the models for accuracy:

1. First, define a helper function `evaluate_model(interpreter)`, which takes model (called `interpreter`) in this case:

```
# A helper function to evaluate the TF Lite model using
"test" dataset.
def evaluate_model(interpreter):
 input_index = interpreter.get_input_details()[0]["index"]
 output_index = interpreter.get_output_details()[0]
 ["index"]
 # Run predictions on every image in the "test" dataset.
 prediction_digits = []
 for test_image in test_images:
  # Pre-processing: add batch dimension and convert to
  float32
  # to match with
  # the model's input data format.
  test_image = np.expand_dims(test_image,
  axis=0).astype(np. float32)
  interpreter.set_tensor(input_index, test_image)
  # Run inference.
  interpreter.invoke()
  # Post-processing: remove batch dimension and find the
  digit with
# highest
  # probability.
  output = interpreter.tensor(output_index)
  digit = np.argmax(output()[0])
  prediction_digits.append(digit)
 accurate_count = 0
 for index in range(len(prediction_digits)):
  if prediction_digits[index] == test_labels[index]:
   accurate_count += 1
 accuracy = accurate_count * 1.0 / len(prediction_digits)
 return accuracy
```

2. Next, we will compare prediction results with ground truth labels to calculate accuracy by calling the `evaluate_model(..)` function defined previously:

```
print(evaluate_model(interpreter))
```

```
0.8499
```

3. Repeat the evaluation on the dynamic range quantized model to obtain:

   ```
   print(evaluate_model(interpreter_quant))
   0.8499
   ```

In this example, the compressed model has no difference at all in the accuracy.

Weight pruning

In *magnitude-based weight pruning model*, sparsity is achieved by gradually zeroing out model weights without compromising accuracy. It helps improve model compression by a factor of *3*. Let us look at a basic sample using the API:

1. Install and setup:

   ```
   import tensorflow as tf
   import numpy as np
   import tensorflow_model_optimization as tfmot
   ```

2. We are going to use MNIST dataset:

   ```
   mnist = tf.keras.datasets.mnist
   (train_images, train_labels), (test_images, test_labels) =
   mnist.load_data()
   # Normalize the input image so that each pixel value is
   between 0 and 1.
   train_images = train_images / 255.0
   test_images = test_images / 255.0
   ```

3. Define the model architecture:

   ```
   # Define the model architecture.
   model_mnist = keras.Sequential([
    keras.layers.InputLayer(input_shape=(28, 28)),
    keras.layers.Reshape(target_shape=(28, 28, 1)),
    keras.layers.Conv2D(filters=12, kernel_size=(3, 3),
    activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(10)
   ```

```
])
```

4. Train the model:

```
# Train the digit classification model
model_mnist.compile(optimizer='adam',
      loss=tf.keras.losses.SparseCategoricalCrossentropy(
      from_logits=True),
      metrics=['accuracy'])
model_mnist.fit(
 train_images,
 train_labels,
 epochs=4, validation_split=0.1,
)
```

It will give accuracy of up to 0.983:

```
Epoch 1/4
1688/1688 [==============================] - 19s 11ms/step
- loss: 0.3288 - accuracy: 0.9063 - val_loss: 0.1459 -
val_accuracy: 0.9608
…
Epoch 4/4
1688/1688 [==============================] - 18s 11ms/step
- loss: 0.0745 - accuracy: 0.9782 - val_loss: 0.0672 -
val_accuracy: 0.9833
```

5. Prune all the weights using tfmot.sparsity.keras.prune_low_magnitude:

```
_, pretrained_weights = tempfile.mkstemp('.tf')
model_mnist.save_weights(pretrained_weights)
model_for_pruning =
tfmot.sparsity.keras.prune_low_magnitude(model_mnist)
model_for_pruning.summary()
_, pretrained_weights = tempfile.mkstemp('.tf')
model_mnist.save_weights(pretrained_weights)
model_for_pruning =
tfmot.sparsity.keras.prune_low_magnitude(model_mnist)
model_for_pruning.summary()
```

6. This function first identifies the weights to be pruned, and then turns them to zero:

```
Model: "sequential_3"
__________________________________________________________
_______
Layer (type) Output Shape Param #
==========================================================
=======
prune_low_magnitude_reshape_ (None, 28, 28, 1) 1
__________________________________________________________
_______
prune_low_magnitude_conv2d_3 (None, 26, 26, 12) 230
__________________________________________________________
_______
prune_low_magnitude_max_pool (None, 13, 13, 12) 1
__________________________________________________________
_______
prune_low_magnitude_flatten_ (None, 2028) 1
__________________________________________________________
_______
prune_low_magnitude_dense_3 (None, 10) 40572
==========================================================
=======
Total params: 40,805
Trainable params: 20,410
Non-trainable params: 20,395
```

Notice that all the layers have `prune_low_magnitude` prefixed.

Pruning some of the layers

Let us look at an example where we prune only dense layers.

It is generally better to fine-tune with pruning as opposed to training from scratch. Try pruning the later layers instead of the first layers. Avoid pruning critical layers (for example, attention mechanism).

The `tfmot.sparsity.keras.prune_low_magnitude` API docs provide details on how to vary the pruning configuration per layer:

```
# Helper function uses `prune_low_magnitude` to make only the
# Dense layers train with pruning.
```

```
def apply_pruning_to_dense(layer):
 if isinstance(layer, tf.keras.layers.Dense):
 return tfmot.sparsity.keras.prune_low_magnitude(layer)
 return layer
# Use `tf.keras.models.clone_model` to apply
`apply_pruning_to_dense`
# to the layers of the model.
model_for_pruning = tf.keras.models.clone_model(
 model_mnist,
 clone_function=apply_pruning_to_dense,
)
model_for_pruning.summary()
```

The following output shows that only **dense_3** layer is pruned:

```
Model: "sequential_3"
______________________________________________________________
___
Layer (type) Output Shape Param #
==============================================================
===
reshape_3 (Reshape) (None, 28, 28, 1) 0
______________________________________________________________
___
conv2d_3 (Conv2D) (None, 26, 26, 12) 120
______________________________________________________________
___
max_pooling2d_3 (MaxPooling2 (None, 13, 13, 12) 0
______________________________________________________________
___
flatten_3 (Flatten) (None, 2028) 0
______________________________________________________________
___
prune_low_magnitude_dense_3 (None, 10) 40572
==============================================================
===
Total params: 40,692
Trainable params: 20,410
Non-trainable params: 20,282
```

Pruning using sequential APIs

Next, we look at the sequential API example for pruning example. In the following example, we will wrap the dense layer in **tfmot.sparsity.keras.prune_low_magnitude**(..) function:

```
# Use `prune_low_magnitude` to make the `Dense` layer train
with pruning.
Input_shape = [10]
model_for_pruning = tf.keras.Sequential([
  keras.layers.InputLayer(input_shape=(28, 28)),
  keras.layers.Reshape(target_shape=(28, 28, 1)),
  keras.layers.Conv2D(filters=12, kernel_size=(3, 3),
  activation=’relu’),
  keras.layers.MaxPooling2D(pool_size=(2, 2)),
  keras.layers.Flatten(),
  tfmot.sparsity.keras.prune_low_magnitude(keras.layers.Dense(1
  0))
])
model_for_pruning.summary()
```

With this, we come to the end of this section on pruning the weights.

Sample can be found at https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch09/post_training_quant_fashion_mnist.ipynb

Weight clustering

Since weights are represented numerically using floating points, **clustering** (https://www.machinecurve.com/index.php/2020/04/23/how-to-perform-mean-shift-clustering-with-python-in-scikit/) techniques can be applied to them in order to identify groups/clusters of similar weights. This is how weight clustering for model optimization works. By applying a clustering technique, the number of unique weights that are present in a machine learning model are reduced.

First, we need a trained model. Applying weight clustering-based optimization to the model involves grouping the weights of layers into *N* clusters, where *N* is configurable. This is performed using some clustering algorithm like **KMeans**.

Once clusters have been computed, all weights in the cluster become the cluster's centroid value. This helps in model compression: values that are equal can be compressed better. TensorFlow team can get *5X* model compression without losing predictive performance in the machine learning model.

Applying weight clustering-based optimization can therefore be a great addition to a machine learning engineer's toolkit, which should include quantization and **pruning** (**https://www.machinecurve.com/index.php/2020/09/23/tensorflow-model-optimization-an-introduction-to-pruning/**).

Let us look at how weight clustering-based model optimization is implemented in TensorFlow.

Weight clustering in TensorFlow

In this section, we'll look at four components of weight clustering in **TFMOT**:

- `cluster_weights(…)`: used for wrapping regular Keras model with weight clustering wrappers, so that clustering can be done.
- `centroidInitialization`: used for computation of the initial values of the cluster centroids which are then used in weight clustering.
- `strip_clustering(…)`: used for stripping the wrappers off clustering-ready Keras model, to get back to normal un-clustered model.
- `cluster_scope(…)`: used when deserializing weight clustered neural network.

Enabling Cluster weights

A Keras model cannot be weight clustered as it lacks certain functionality for doing so. That is why we will need to wrap the model with the functionality, which clusters weights during training. It is the way to configure weight clustering for the Keras model.

One should only cluster a model that already shows acceptable accuracy or other relevant metrics.

Applying **`cluster_weights(...)`** works shown as follows (*source*: **TensorFlow** license (**https://www.tensorflow.org/model_optimization/api_docs/python/tfmot/clustering/keras/cluster_weights**): **Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/**):

```
clustering_params = {
  'number_of_clusters': 8,
  'cluster_centroids_init':
     CentroidInitialization.DENSITY_BASED
}
clustered_model = cluster_weights(original_model,
**clustering_params)
```

In this code, we define the number of clusters we want (we will compare *4* versus *8* clusters in a sample later), as well as how the centroids are initialized – a configuration option that we will look at in more detail next. Subsequently, we are passing the clustering parameters into **`cluster_weights(...)`** without the original model. The **`clustered_model`** that is returned can then be used for clustering.

Determining centroid initialization: CentroidInitialization

From the previous section, we know that weight clustering involves clustering the weights but also replacing the weights that are part of a cluster with the centroids of that cluster. This achieves the benefits in terms of compression that we discussed.

Let us explore the clustering techniques supported by TFMOT.

Stripping clustering wrappers

Using **`strip_clustering(…)`** to remove special wrapper created by **`cluster_weights`**:

```
model = tensorflow.keras.Model(...)
wrapped_model = cluster_weights(model)
stripped_model = strip_clustering(wrapped_model)
```

Serializing the clustered model

- In case you need to save the wrapped model:

```
model = tf.Keras.Model(...)
wrapped_model = cluster_weights(model)
tensorflow.keras.models.save_model(wrapped_model,
'./some_path')
```

- This becomes tricky as clustered model cannot be loaded directly. TFMOT provides a technique: **`tfmot.clustering.keras.cluster_scope():`** as follows:

```
model = tf.Keras.Model(...)
wrapped_model = cluster_weights(model)
file_path = './some_path'
tensorflow.keras.models.save_model(wrapped_model,
file_path)
with tfmot.clustering.keras.cluster_scope():
  loaded_model =
  tensorflow.keras.models.load_model(file_path)
```

Weight clustering MNIST classification model

Let us use weight clustering to look at the accuracy of the model. We will look at the end to end sample code for the same. Assuming the TensorFlow quantization toolkit is already installed:

```
import tensorflowas tf
import numpy as np
import tempfile
import zipfile
import os
```

Let us first train the model without clustering:

1. Load the dataset.
2. Train and test images and normalize.
3. Create sequential model.
4. Compile the model with following parameters:
    a. Use **`adam`** optimizer.
    b. **`SparseCategoricalCrossentropy`**
    c. Optimize for **`accuracy`** metrics.

5. Run **`model.fit(..)`** with **`train_images`** and **`train_labels`** for **`10`** epochs and validation split of **`0.1`**:

```
# Load MNIST dataset
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
# Normalize the input image so that each pixel value is
between 0 to 1.
train_images = train_images / 255.0
test_images = test_images / 255.0
model = tf.keras.Sequential([
 tf.keras.layers.InputLayer(input_shape=(28, 28)),
 tf.keras.layers.Reshape(target_shape=(28, 28, 1)),
 tf.keras.layers.Conv2D(filters=12, kernel_size=(3, 3),
        activation=tf.nn.relu),
 tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
 tf.keras.layers.Flatten(),
 tf.keras.layers.Dense(10)
])
# Train the digit classification model
model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_lo
gits=True), metrics=['accuracy'])
model.fit(
  train_images,
  train_labels,
  validation_split=0.1,
  epochs=10
)
```

The output will show the accuracy and loss for training and validation set. We will compare the validation accuracy we got, as follows (**`0.9842`**), which we will compare later:

```
Epoch 1/10
1688/1688 [==============================] - 18s 10ms/step
- loss: 0.3037 - accuracy: 0.9137 - val_loss: 0.1267 -
val_accuracy: 0.9680
..
```

```
Epoch 10/10
1688/1688 [==============================] - 17s 10ms/step
- loss: 0.0355 - accuracy: 0.9896 - val_loss: 0.0634 -
val_accuracy: 0.9842
```

Baseline accuracy:

```
_, baseline_model_accuracy = model.evaluate(
  test_images, test_labels, verbose=0)
print('Baseline test accuracy:', baseline_model_accuracy)
_, keras_file = tempfile.mkstemp('.h5')
print('Saving model to: ', keras_file)
tf.keras.models.save_model(model, keras_file,
include_optimizer=False)
Baseline test accuracy: 0.9824000000953674
Saving model to: /tmp/tmp6feo6no6.h5
```

Cluster the weight with 8 clusters

Apply the **`cluster_weights()`** API to cluster the whole pre-trained model to demonstrate and observe its effectiveness in reducing the model size when applying ZIP, while maintaining accuracy. For more details refer to the **clustering comprehensive guide** (**https://www.tensorflow.org/model_optimization/guide/clustering/clustering_comprehensive_guide**).

Define the model and apply the clustering API

The model needs to be pre-trained before using the clustering API:

1. This function wraps a Keras model or layer with clustering functionality, which clusters the layer's weights during training. For example, using this with **`number_of_clusters`** equals to **`8`** will ensure that each weight tensor has no more than **`8`** unique values:

```
import tensorflow_model_optimization as tfmot
cluster_weights = tfmot.clustering.keras.cluster_weights
CentroidInitialization =
tfmot.clustering.keras.CentroidInitialization
clustering_params = {
 'number_of_clusters': 8,
 'cluster_centroids_init':
 CentroidInitialization.KMEANS_PLUS_PLUS
}
clustered_model = cluster_weights(model,
**clustering_params)
# Use smaller learning rate for fine-tuning
opt = tf.keras.optimizers.Adam(learning_rate=1e-5)
clustered_model.compile(
 loss=tf.keras.losses.SparseCategoricalCrossentropy(from_l
 ogits=True),
 optimizer=opt,
 metrics=['accuracy'])
clustered_model.summary()
```

Summary output will show the clustered layers:

```
Model: "sequential"
__________________________________________________________
_______
Layer (type) Output Shape Param #
==========================================================
=======
cluster_reshape (ClusterWeight (None, 28, 28, 1) 0
__________________________________________________________
_______
cluster_conv2d (ClusterWeight (None, 26, 26, 12) 236
__________________________________________________________
_______
cluster_max_pooling2d (Clust.. (None, 13, 13, 12) 0
__________________________________________________________
_______
cluster_flatten (ClusterWeig.. (None, 2028) 0
__________________________________________________________
_______
cluster_dense (ClusterWeight.. (None, 10) 40578
==========================================================
=======
Total params: 40,814
Trainable params: 20,426
```

```
Non-trainable params: 20,388
```

2. Fine-tune the clustered model:

```
# Fine-tune model
clustered_model.fit( train_images, train_labels, epochs=3,
validation_split=0.1)
Epoch 1/3 1688/1688 [==============================] - 22s
13ms/step - loss: 0.0494 - accuracy: 0.9843 - val_loss:
0.0648 - val_accuracy: 0.9837 Epoch 2/3 1688/1688
[==============================] - 21s 13ms/step - loss:
0.0361 - accuracy: 0.9887 - val_loss: 0.0614 -
val_accuracy: 0.9843 Epoch 3/3 1688/1688
[==============================] - 21s 12ms/step - loss:
0.0340 - accuracy: 0.9895 - val_loss: 0.0595 -
val_accuracy: 0.9848
```

3. Next, let us find the number of clusters by defining a *helper* function:

```
def print_model_weight_clusters(model):
  for layer in model.layers:
   if isinstance(layer, tf.keras.layers.Wrapper):
     weights = layer.trainable_weights
   else:
     weights = layer.weights
   for weight in weights:
     # ignore auxiliary quantization weights
     if "quantize_layer" in weight.name:
       continue
     if "kernel" in weight.name:
       unique_count = len(np.unique(weight))
       print(
       f"{layer.name}/{weight.name}: {unique_count}
       clusters "
       )
stripped_clustered_model =
 tfmot.clustering.keras.strip_clustering(clustered_model)
print_model_weight_clusters(stripped_clustered_model)
```

Print output is listed as follows, which shows clusters for **conv2d** and dense layers:

```
conv2d/kernel:0: 8 clusters
dense/kernel:0: 8 clusters
```

4. Evaluate the clustering model with the baseline model on validation accuracy:

```
_, clustered_model_accuracy = clustered_model.evaluate(
test_images, test_labels, verbose=0)
print('Baseline test accuracy:', baseline_model_accuracy)
print('Clustered test accuracy:',
clustered_model_accuracy)
Baseline test accuracy: 0.9824000000953674
Clustered test accuracy: 0.9797000288963318
```

As can be seen, the model accuracy is not reduced significantly.

Compare the size of a normal model, a clustered model, and a clustered TFLite model

Let us compare the clustered model with normal model, in terms of its size:

```
final_model =
tfmot.clustering.keras.strip_clustering(clustered_model)
_, clustered_keras_file = tempfile.mkstemp('.h5')
print('Saving clustered model to: ', clustered_keras_file)
tf.keras.models.save_model(final_model, clustered_keras_file,
        include_optimizer=False)
Create and save TFLite model from clustered model:
clustered_tflite_file = '/tmp/clustered_mnist.tflite'
converter =
tf.lite.TFLiteConverter.from_keras_model(final_model)
tflite_clustered_model = converter.convert()
with open(clustered_tflite_file, 'wb') as f:
f.write(tflite_clustered_model)
print('Saved clustered TFLite model to:',
clustered_tflite_file)
INFO:tensorflow:Assets written to: /tmp/tmpfmsd8iy5/assets
INFO:tensorflow:Assets written to: /tmp/tmpfmsd8iy5/assets
Saved clustered TFLite model to: /tmp/clustered_mnist.tflite
```

Helper function to ZIP the file:

```
def get_gzipped_model_size(file):
# It returns the size of the gzipped model in bytes.
   import os
   import zipfile
_, zipped_file = tempfile.mkstemp('.zip')
   with zipfile.ZipFile(zipped_file, 'w',
   compression=zipfile.ZIP_DEFLATED) as f:
   f.write(file)
   return os.path.getsize(zipped_file)
```

Compare the models: normal baseline model, clustered model, and clustered TFLite model:

```
print("Size of gzipped baseline Keras model: %.2f bytes" %
(get_gzipped_model_size(keras_file)))
print("Size of gzipped clustered Keras model: %.2f bytes" %
(get_gzipped_model_size(clustered_keras_file)))
print("Size of gzipped clustered TFlite model: %.2f bytes" %
(get_gzipped_model_size(clustered_tflite_file)))
```

```
Size of gzipped baseline Keras model: 78280.00 bytes
Size of gzipped clustered Keras model: 13414.00 bytes
Size of gzipped clustered TFlite model: 12969.00 bytes
```

As you can see, the clustered model is compressed by *83%,* and is almost *17%* of the original model in size.

Conclusion

In this chapter, we learnt the concept of model optimization in neural networks, and how it has been implemented in TensorFlow. We explored techniques like quantization, both while training and post training. We looked at an *end-to-end usecase* with Fashion MNIST dataset. We also looked at weight clustering technique to reduce the model size.

Questions

1. **Pruning can be applied to some of the layers or all the layers in the model:**

   a. true
   b. false

2. **Quantization is part core TensorFlow installation.**

   a. true
   b. false

3. **Weight clustering can help reduce the model size up to:**

   a. 0x
   b. 2x
   c. 100x
   d. 8x

Answers

1. **a**
2. **b**
3. **d**

References

- **TensorFlow model optimization: quantization and pruning (TF World '19)**: **https://www.youtube.com/watch?v=3JWRVx1OKQQ**.
- **Quantization for neural networks**: **https://leimao.github.io/article/Neural-Networks-Quantization/**.
- **TensorFlow model optimization: an introduction to quantization**: **https://www.machinecurve.com/index.php/2020/09/16/tensorflow-model-optimization-an-introduction-to-quantization/**.
- **Weight clustering comprehensive guide**: **https://www.tensorflow.org/model_optimization/guide/clustering/clustering_comprehensive_guide**.

CHAPTER 10
Generative Adversarial Networks

Introduction

In this chapter, we will understand the concept behind **GANs,** also called **General Adversarial Networks**. A generative adversarial network design by *Ian Goodfellow* and his colleagues, it is made up of two neural networks in a contest with each other in the form of a zero-sum game, where one agent's gain becomes another agent's loss.

For a data set, this technique learns to generate new data with the same statistics as the training set. It could be used to generate new photographs from the existing dataset of photographs. It has proved useful for **semi-supervised**, **fully supervised**, and **reinforcement learning**.

The idea behind GAN is of indirect training through discriminator, which is itself being updated dynamically. The generator is being trained to fool the discriminator. This enables the model to learn using an unsupervised technique.

We look at the various approaches to make the generators more effective at creating more realistic images. We will explore the three GAN topologies: **simple GAN**, **DCGAN**, and **WGAN**.

Structure

In this chapter, we will cover the following topics:

- Discriminator
- Generator
- Loss function
- Simple GAN
- DCGAN
- WGAN

Objective

In this chapter, we will learn the basics of GAN and its variations, like DCGAN and WGAN. We will use them to generate fake images for **FASHION MNIST** dataset.

Introducing GANs

There are two types of machine learning techniques: **d**iscriminative models and **g**enerative models. In the last few chapters, we have covered discriminative models that use deep learning to classify into two or more classes or predict a continuous variable.

In discriminative models, if given *X*, we predict *Y*:

X Š *Y*

P(X|Y)

In this chapter, we are going to learn generative models. In generative models, if given *Y* + some noise, the model predicts a vector of features in a latent space:

x, *Y* Š *X*

Here, noise plays an important role in generating different kinds of realistic pictures, given the class label. Generative models capture probabilistic distribution of features of *X*.

Generative models are of two types: variational auto encoders and generative adversarial networks. In variational autoencoders, the input is mapped to a latent space by an encoder, and then decoded back by the decoder as shown in *figure 10.1*:

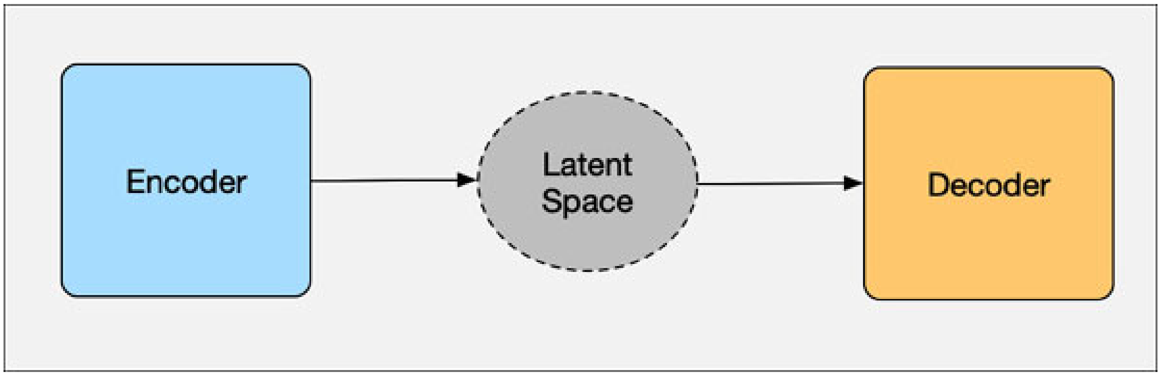


Figure 10.1: *Variational autoencoders*

The encoder takes in a realistic image and the decoder outputs an image from the input of vectors. The variation part in the variation encoders adds some noise to the input.

In generative adversarial networks or GANs, there is a generator instead of a decoder. It takes a random vector and generates a fake image. This fake image and real image are fed into the discriminator to compare and provide feedback into the generator as shown in *figure 10.2*:

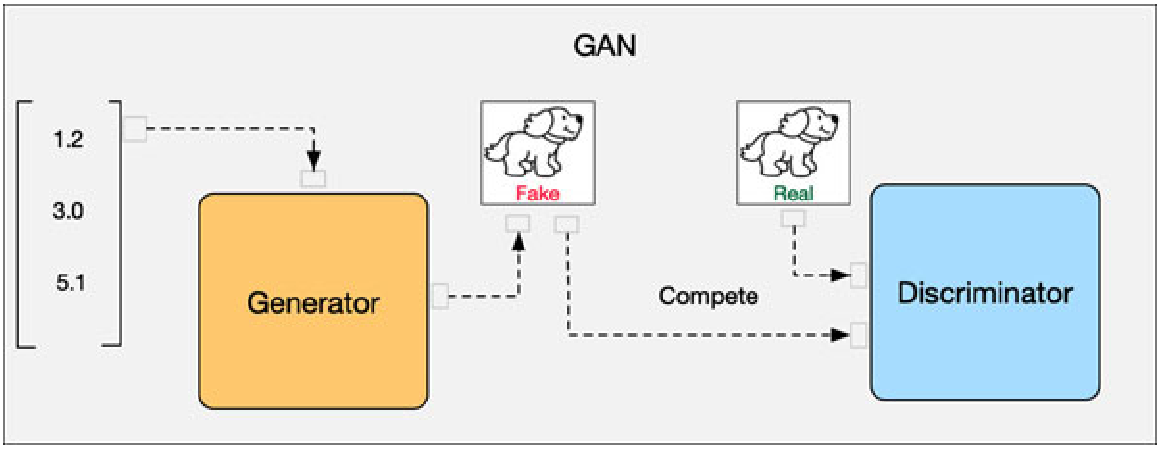


Figure 10.2: *Various components of a GAN – generator and discriminator*

Once the generator is smart enough, it can start producing realistic real-life images from random inputs. *Figure 10.3* shows the nomenclature for the inputs and outputs of generator and discriminator:

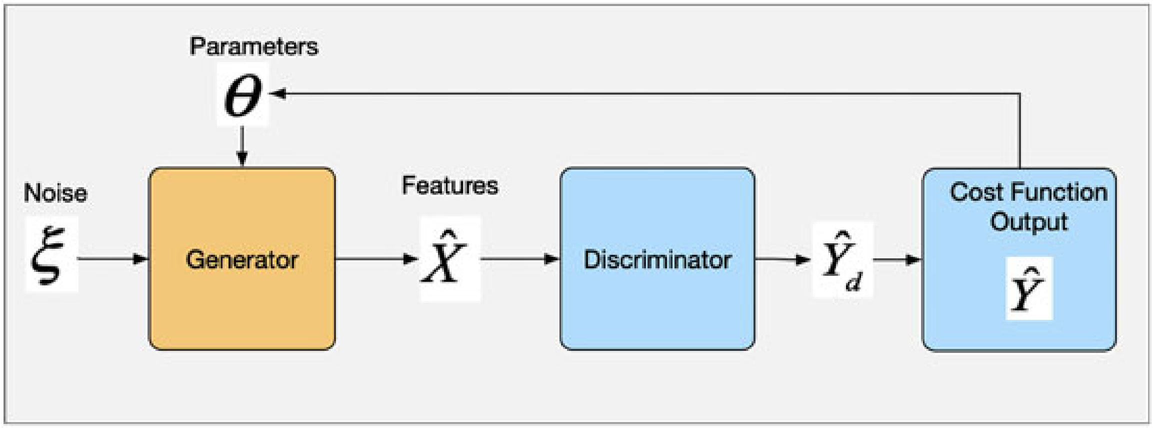


Figure 10.3: *Generator's input, output and parameters*

Generator takes noise and output features $\hat{X}$. These are tested by discriminator and the output is $\hat{Y}_d$. These are fed into a cost function and the parameters q are adjusted.

Discriminator

Let us start by looking at the function or classifier. For example, an image classifier helps determine the label of an image with a probability. The following *figure 10.4* shows a classifier classifying fruits.

Classifiers work on features that are fed as vectors in the neural network:

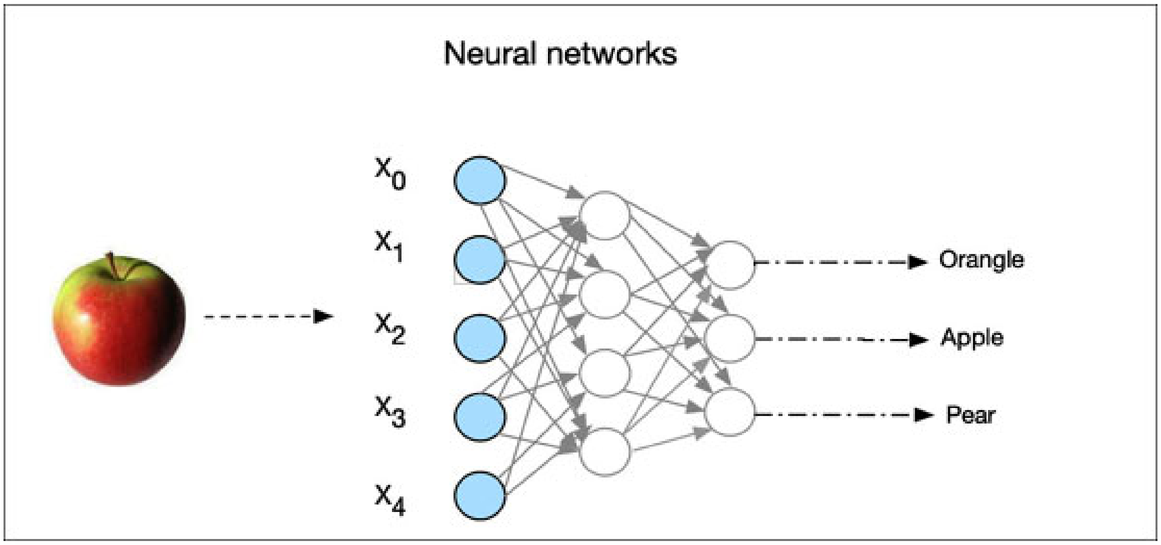


Figure 10.4: *Neural network input layer*

In the following *figure 10.5*, x0 to x4 are fed into the input layer:

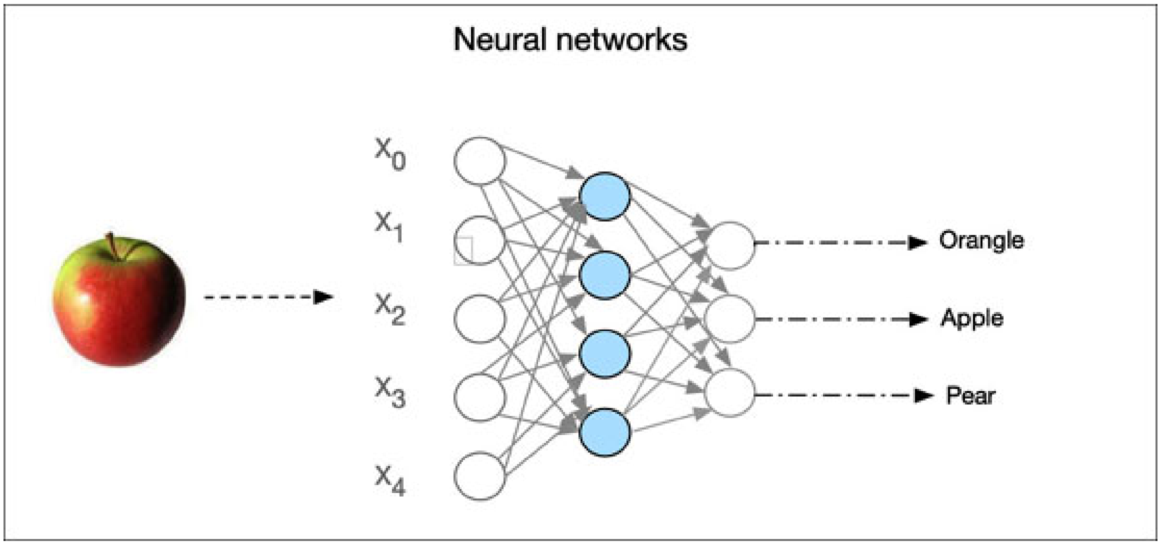


Figure 10.5: *Neural network input layer*

Neural networks then capture these non-linearities and outputs the probabilities as shown in *figure 10.6*:

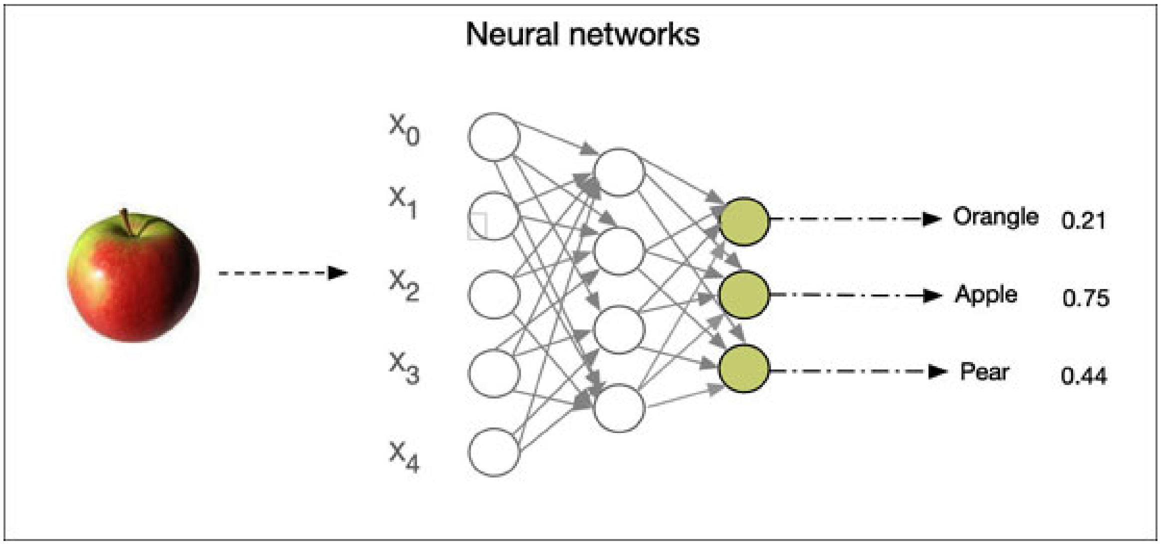


Figure 10.6: *Neural network output layer: predicting labels and their probabilities*

The last layer outputs the class label and probabilities. Given *X*, the feature model predicts probability of *Y*.

Model is parameter $P(Y| X)$ ŠΘ. Predicted values $\hat{Y}$ are compared to Y in order to fine-tune the hyperparameters Θ. *Figure 10.7* shows the schematics of a discriminator. It takes an image as an input and provides an output whether it is fake with a probability:

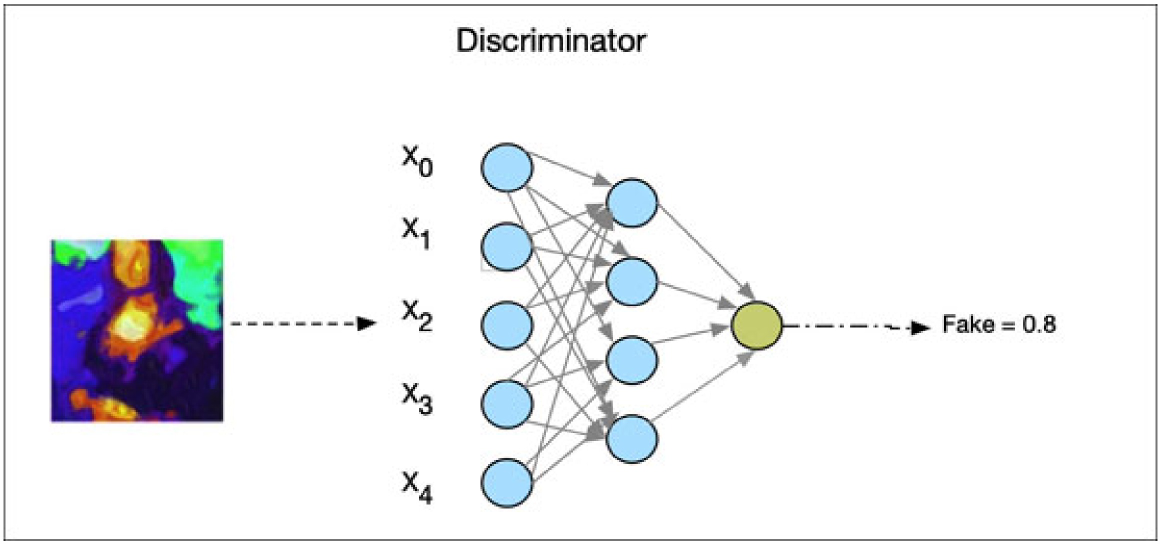


Figure 10.7: *Neural network output layer: predicting labels and their probabilities*

Discriminator is a type of classifier which calculates probability of an image being a fake. These probabilities are then fed to the generator.

Generator

The generator is the model that generates the examples as shown in the *figure 10.8*. We try to make it better by training:

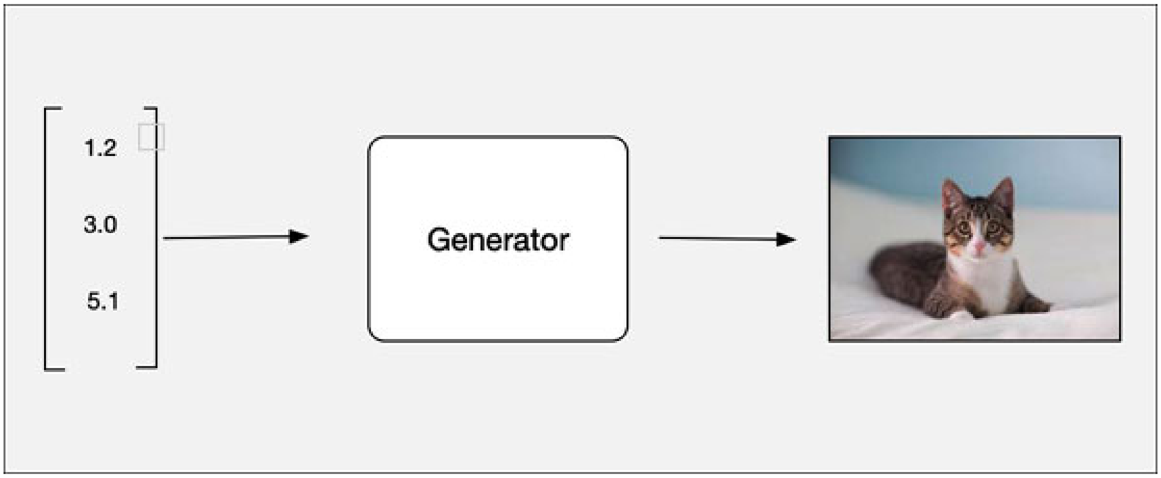


Figure 10.8: *Generator – input and output*

We will explore how the generator improves its performance. If the generator is trained to produce cats after each run, it will do so, and should ideally produce a different cat every time. To make sure this happens, a different noise vector must be input as shown in *figure 10.9*:

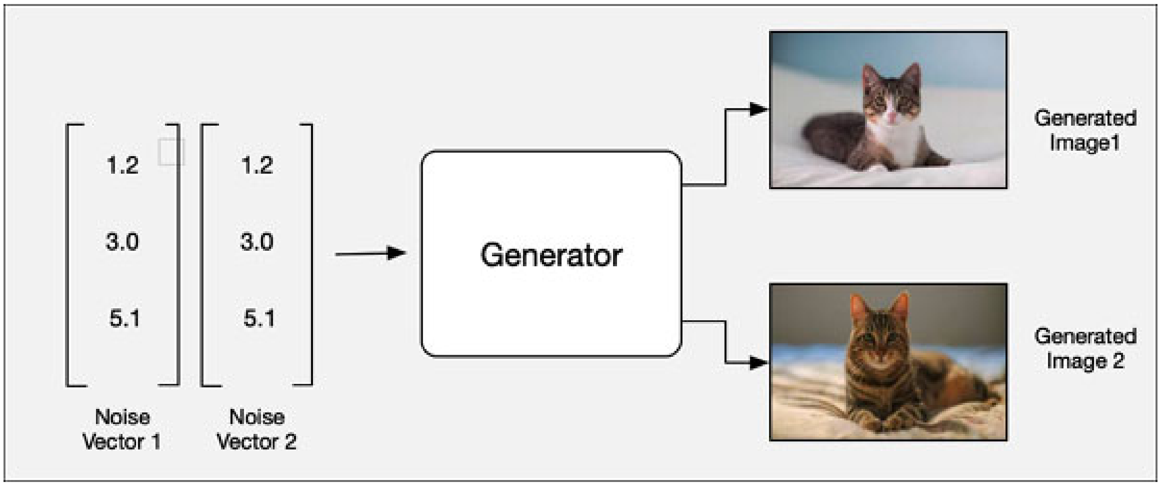


Figure 10.9: *Different noise vector inputs to generator outputs different generated images*

We looked at the basics of generator. Now, let us look at the cost function that helps improve its accuracy.

BCE cost function

Let us look at the cost function, which we are trying to reduce to a minimum. The following equation is the cost function with all the relevant terms:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h\left(x^{(i)},\theta\right)+\left(1-y^{(i)}\right)\log\left(1-h\left(x^{(i)},\theta\right)\right)\right]$$

Equation 10.1: *Cost function of the generator*

- *m:* batch size
- *J(θ):* average loss of the whole batch
- *h* : predictions made by the model
- $y^{(i)}$: true labels
- $x^{(i)}$: features passed through predictions
- θ: parameters of the generator

Let us break down the cost function into subsets:

$y^{(i)}\log h\left(x^{(i)},\theta\right)$ is a multiplication of true labels and log of prediction labels given $x^{(i)}$ and θ.

Table 10.1 describes the interpretation of the first part of the equation:

A	B	C
$y^{(i)}$	$h(x^{(i)}, \theta)$	$y^{(i)}\log h\left(x^{(i)},\theta\right)$
0	Any	0
1	~0.99	0
1	~0	-infinity

Table 10.1: *Interpreting first part of the BCE cost function*

If *y* is *0,* irrespective of the predicted value, column **C** is always *0*. If both *y* and the predicted value is near *1*, column **C** becomes approaching *0*. If *y* is *1* and the predicted value nears *0*, then it means the error is large and the third column C becomes a very large negative number (approaching infinity as *h* approaches *0*).

$(1-y^{(i)})$ $\log(1-h(x^{(i)}\theta))$ is a multiplication of wrong value and log of predicted wrong value.

The following *table 10.2* shows how this new value fares with different values of A and B:

A	**B**	**C**
$y^{(i)}$	$h(x^{(i)}, \theta)$	$1 - y^{(i)} \log h\left(x^{(i)}, \theta\right)$
1	Any	0
0	~0.99	-infinity
0	~0	0

Table 10.2: *Interpretation of second part of the equation*

The term in column **B** seems to become relevant when *y* is *0,* and column **C** must highlight values which are far away from prediction, for example, with a value of **0.99** the predicted value column **C** is -infinity.

Let us plot the two terms as shown in the *figure 10.10*:

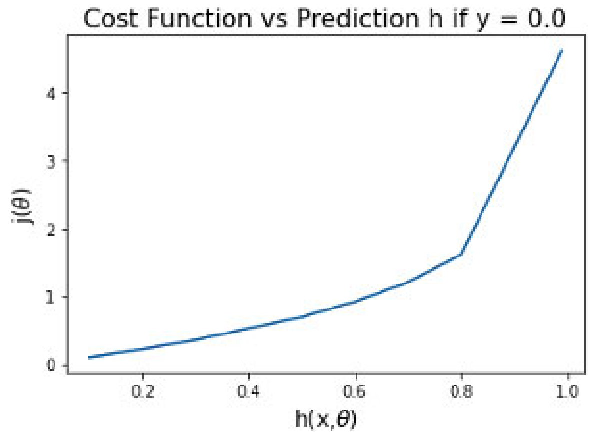


Figure 10.10: *Cost function as a function of prediction if y is 0.0, that is, image is fake*

Let us plot the cost function *J*(θ) as a prediction *h,* if *y = 1.0* as shown in the *figure 10.11*.

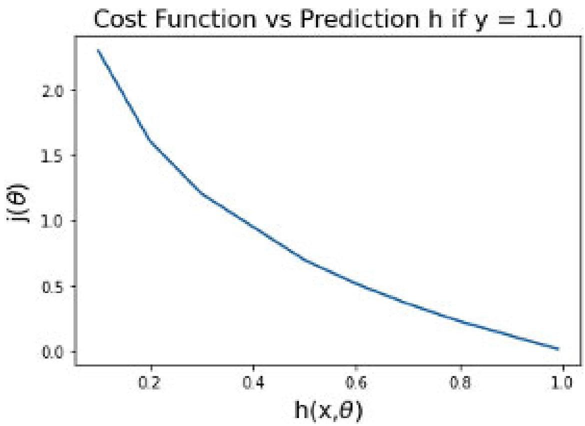


Figure 10.11: *Cost function as a function of prediction if y is 1.0, that is, image is real*

To summarize, BCE cost function has two parts. It is close to zero when the prediction is close to one the other part.

GAN for FASHION MNIST images

In this section, we are going to look at a GAN which generates new images looking at the MNIST images:

1. Import the classes and packages:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
BATCH_SIZE = 256
BUFFER_SIZE = 60000
EPOCHES = 300
OUTPUT_DIR = "img_fashionmnist"
```

2. Load the dataset:

```
fmnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
fmnist.load_data()
Downloading data from
https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
     32768/29515 [=================================] - 0s
     0us/step
```

```
Downloading data from
https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s
0us/step
Downloading data from
https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148
[================================================] - 0s
0us/step
Downloading data from
https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s
0us/step
```

3. Let us look at the shape of the training data:

```
(train_images[0].shape)
```

The output will be **(28,28)**. Let us plot the first image from the training dataset:

```
plt.imshow(train_images[1], cmap = "gray")
```

We will get the output as shown in *figure 10.12*:

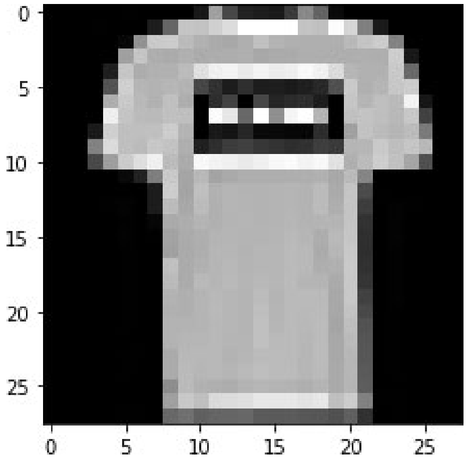


Figure 10.12: *Real image sample*

4. Convert the datatype for training images and normalize it:

```
train_images = train_images.astype("float32")
train_images = (train_images - 127.5) / 127.5
```

5. Reshape the data set to (60000, 784):

```
train_dataset =
tf.data.Dataset.from_tensor_slices(train_images.reshape(trai
n_images.shape[0],784)).shuffle(BUFFER_SIZE).batch(BATCH_SIZ
E)
```

Next, let us create a basic generator.

Generator

Subclass of keras.Model, it is initialized with the various dense and activation functions. In the call method, it takes random noise and converts it into a dense layer of 784 units using three dense layers and LeakyReLU:

```
class Generator(keras.Model):
  def __init__(self, random_noise_size = 100):
    super().__init__(name='generator')
    #layers
      self.input_layer = keras.layers.Dense(units =
      random_noise_size)
      self.dense_1 = keras.layers.Dense(units = 128)
      self.leaky_1 = keras.layers.LeakyReLU(alpha = 0.01)
      self.dense_2 = keras.layers.Dense(units = 128)
      self.leaky_2 = keras.layers.LeakyReLU(alpha = 0.01)
      self.dense_3 = keras.layers.Dense(units = 256)
      self.leaky_3 = keras.layers.LeakyReLU(alpha = 0.01)
      self.output_layer = keras.layers.Dense(units=784,
      activation = "tanh")
  def call(self, input_tensor):
    ## Definition of Forward Pass
    x = self.input_layer(input_tensor)
    x = self.dense_1(x)
    x = self.leaky_1(x)
    x = self.dense_2(x)
    x = self.leaky_2(x)
    x = self.dense_3(x)
    x = self.leaky_3(x)
    return self.output_layer(x)
  def generate_noise(self,batch_size, random_noise_size):
    return np.random.uniform(-1,1, size = (batch_size,
 random_noise_size))
```

Objective function

We will use **`BinaryCrossEntropy`** as the activation function:

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits =
True)
```

The following equation describes the cross-entropy equation in more detail:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

Where,

- $J(\theta)$: loss function
- $y^{(i)}$: predicted labels
- $x^{(i)}$: features
- θ: parameter

Let us define the generator objective function:

```
def generator_objective(dx_of_gx):
    # Labels are true here because generator thinks
# he produces real images.
    return cross_entropy(tf.ones_like(dx_of_gx), dx_of_gx)
```

Let us plot the fake image created by the generator by create a fake image and name it fake_image from the generator:

```
generator = Generator()
fake_image = generator(np.random.uniform(-1,1, size =(1,100)))
fake_image = tf.reshape(fake_image, shape = (28,28))
plt.imshow(fake_image, cmap = "gray")
```

Notice that the image is totally random with *28x28* pixel and one channel (that is, it is grey scale) as shown in *figure 10.13*:

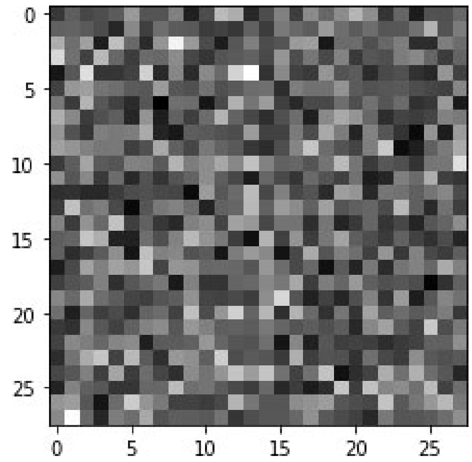


Figure 10.13: *Fake image plotted*

Next, let us look at the discriminator and its implementation using TensorFlow 2.x.

Discriminator

Discriminator takes input of reshaped image *28x28* pixel into 784 and takes it through a combination of three dense and LeakyReLU layers before the final dense layer which outputs the shape of one-dimensional tensor:

```
class Discriminator(keras.Model):
  def __init__(self):
    super().__init__(name = "discriminator")
    #Layers
    self.input_layer = keras.layers.Dense(units = 784)
    self.dense_1 = keras.layers.Dense(units = 128)
    self.leaky_1 = keras.layers.LeakyReLU(alpha = 0.01)
    self.dense_2 = keras.layers.Dense(units = 128)
    self.leaky_2 = keras.layers.LeakyReLU(alpha = 0.01)
    self.dense_3 = keras.layers.Dense(units = 128)
    self.leaky_3 = keras.layers.LeakyReLU(alpha = 0.01)
    self.logits = keras.layers.Dense(units = 1) # This neuron tells us if
    the input is fake or real
  def call(self, input_tensor):
     ## Definition of Forward Pass
    x = self.input_layer(input_tensor)
    x = self.dense_1(x)
    x = self.leaky_1(x)
    x = self.leaky_2(x)
    x = self.leaky_3(x)
    x = self.leaky_3(x)
    x = self.logits(x)
    return x
```

We will initialize the discriminator created earlier:

```
discriminator = Discriminator()
```

Our objective function is a combination of real and fake loss as we saw earlier:

```
def discriminator_objective(d_x, g_z, smoothing_factor = 0.9):
  """
  d_x = real output
  g_z = fake output
  """
    real_loss = cross_entropy(tf.ones_like(d_x) *
    smoothing_factor, d_x)
  # If we feed the discriminator with real images, we assume they
  all are
  # the right pictures --> Because of that label == 1
    fake_loss = cross_entropy(tf.zeros_like(g_z), g_z)
    # Each noise we feed in are fakes image -->
    # Because of that labels are 0
    total_loss = real_loss + fake_loss
  return total_loss
```

Next, we define the optimizer:

```
generator_optimizer = keras.optimizers.RMSprop()
discriminator_optimizer = keras.optimizers.RMSprop()
```

We are using RMSprop as an optimization algorithm.

RMSprop is an unpublished optimization algorithm designed for neural networks. It was first proposed by *Geoff Hinton* in *Lecture 6* of the online course *Neural Networks for Machine Learning* (*https://www.coursera.org/learn/neural-networks/home/welcome*).

RMSprop is an adaptive learning rate method, which has been growing in popularity in the recent years. It is famous for not being published, yet being very well-known; most deep learning frameworks include the implementation of it out of the box.

Training function

We have defined a tf.function which takes discriminator and generator for each batch and calculates the loss:

```
@tf.function()
def training_step(generator: Discriminator, discriminator:
Discriminator, images:np.ndarray , k:int =1, batch_size = 32):
 for _ in range(k):
  with tf.GradientTape() as gen_tape, tf.GradientTape() as
  disc_tape:
    noise = generator.generate_noise(batch_size, 100)
    g_z = generator(noise)
    d_x_true = discriminator(images) # Trainable?
    d_x_fake = discriminator(g_z) # dx_of_gx
    discriminator_loss = discriminator_objective(d_x_true,
 d_x_fake)
    # Adjusting Gradient of Discriminator
    gradients_of_discriminator = disc_tape.gradient(
 discriminator_loss, discriminator.trainable_variables)
    discriminator_optimizer.apply_gradients(zip(
 gradients_of_discriminator,
 discriminator.trainable_variables))
 # Takes a list of gradient and variables pairs
    generator_loss = generator_objective(d_x_fake)
    # Adjusting Gradient of Generator
    gradients_of_generator = gen_tape.gradient(generator_loss,
 generator.trainable_variables)
   generator_optimizer.apply_gradients(zip(
 gradients_of_generator, generator.trainable_variables))
```

Next, we define the training function which uses the training step:

```
def training(dataset, epoches):
 for epoch in range(epoches):
  for batch in dataset:
    training_step(generator, discriminator, batch ,
 batch_size = BATCH_SIZE, k = 1)
  ## After ith epoch plot image
  if (epoch % 50) == 0:
    fake_image = tf.reshape(generator(seed), shape = (28,28))
    print("{}/{} epoches".format(epoch, epoches))
```

```
        plt.imsave("{}/{}.png".format(OUTPUT_DIR,epoch),fake_image,
        cmap = "gray")
```

Call the training function at `300` epoches:

```
%%time
training(train_dataset, EPOCHES)
```

It will take around 1 and half hours, approximately.

```
0/300 epoches
50/300 epoches
100/300 epoches
150/300 epoches
200/300 epoches
250/300 epoches
CPU times: user 1h 38min 25s, sys: 3min 3s, total: 1h 41min 28s
```

Let us plot the generated image as shown in *figure 10.14*:

```
fake_image = generator(np.random.uniform(-1,1, size = (1, 100)))
plt.imshow(tf.reshape(fake_image, shape = (28,28)), cmap="gray")
```

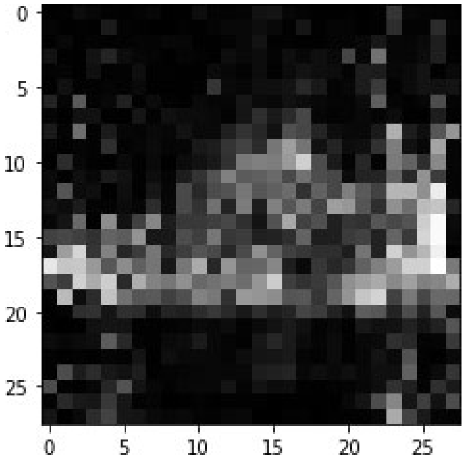


Figure 10.14: *Fake image generated using simple GAN*

In the next section, we look at another technique for generator and discriminator to improve the efficiency and performance of the networks called **Dynamic Convolutional GAN** or **DCGAN**.

DCGAN

Attempts to scale up GANs using CNNs to model images have been unsuccessful. This motivated the authors of *LAPGAN (Denton et al., 2015)* to develop an approach to iteratively upscale low resolution generated images that can be modeled more reliably. They encountered difficulties attempting to scale GANs using CNN architectures used in the supervised literature. However, after extensive model exploration, they identified a family of architecture that resulted in stable training across a range of datasets and it also allowed for training higher resolution and deeper generative models. Core to this approach is adopting and modifying three changes to CNN architectures:

- The first approach is all convolutional net (*Springenberg et al., 2014*) which replaces deterministic spatial pooling functions (max pooling) with strided convolutions, allowing the network to learn its own spatial down sampling. This approach is used by DCGAN in the generator, allowing it to learn its own spatial up sampling and discriminator.
- Second is to eliminate fully connected layers on top of convolutional features. The strongest example of this is global average pooling which has been utilized in *state-of-the-art image* classification models (*Mordvintsev et al.*). Authors found global average pooling increased model stability but hurt the convergence speed.
- Third approach is **b**atch **n**ormalization (*Ioffe & Szegedy, 2015*), which helps stabilize learning by normalizing the input to each unit to have zero mean and unit variance. It helps in dealing with training problems that arise due to poor initialization and helps gradient flow in deeper models. This proved critical to help generators to begin learning and preventing the generator from collapsing all samples to a single point which is a common failure mode observed in GANs. Applying batch normalization to all layers however, resulted in sample oscillation and model instability. This has been avoided by not applying the batch norm to the generator output layer and the discriminator input layer.

  The ReLU activation (*Nair & Hinton, 2010*) has been used in the generator except the output layer which uses the tanh function. Authors observed that using a bounded activation allows the model to learn more quickly to saturate. It also helped cover the color space of the training distribution. Within the discriminator, the authors found the leaky rectified activation (*Maas et al., 2013*) (*Xu et al., 2015*) to work well. This was in contrast to the original GAN paper, which used the maxout activation (*Goodfellow et al., 2013*).

To summarize, the following steps are implemented in DCGAN:

1. CNN replaces max pooling with strided convolutions.

2. It eliminates fully connected layers on top of convoluted features.
3. It uses batch normalization.
4. ReLU activation is used in the generator except the last layer.
5. The last layer of the generator uses `tanh` activation function.

Generator

The generator uses **tf.keras.layers.Conv2DTranspose** (up sampling) layers to produce an image from a seed (random noise). Start with a dense layer that takes this seed as input, then up sample several times until you reach the desired image size of *28x28x1*. Notice the **tf.keras.layers.LeakyReLU** activation for each layer, except the output layer which uses `tanh`:

```
def make_generator_model():
  model = tf.keras.Sequential()
  model.add(layers.Dense(7*7*256, use_bias=False, input_shape=
  (100,)))
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())
  model.add(layers.Reshape((7, 7, 256)))
  assert model.output_shape == (None, 7, 7, 256) # Note: None is
  the batch size
  model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
   padding='same', use_bias=False))
  assert model.output_shape == (None, 7, 7, 128)
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())
  model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
   padding='same', use_bias=False))
  assert model.output_shape == (None, 14, 14, 64)
  model.add(layers.BatchNormalization())
  model.add(layers.LeakyReLU())
  model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
     padding='same', use_bias=False, activation='tanh'))
  assert model.output_shape == (None, 28, 28, 1)
  return model
```

After creating a generator, let us initialize it, and use it to generate an image from noise. Use the generator to create an image:

```
generator = make_generator_model()
```

```
noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)
plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

The output is as shown in the *figure 10.15*:

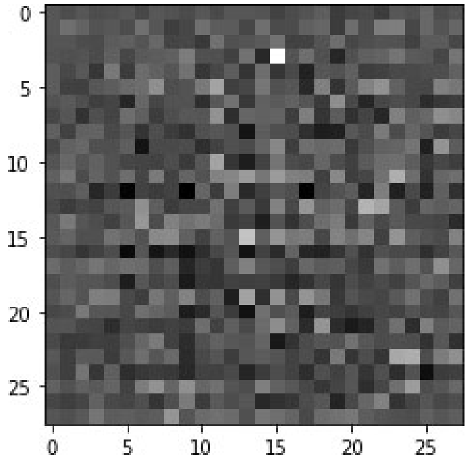


Figure 10.15: *Generated image*

Next, let us look at the discriminator for DCGAN based technique.

Discriminator

Discriminator for DCGAN is a CNN based classifier with LeakyReLU between each Conv2D layer and **d**ropout layer. The last layer is a dense layer with tensor for dimension 1:

```
def make_discriminator_model():
  model = tf.keras.Sequential()
  model.add(layers.Conv2D(64, (5, 5), strides=(2, 2),
  padding='same',
          input_shape=[28, 28, 1]))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.3))
   model.add(layers.Conv2D(128, (5, 5), strides=(2, 2),
   padding='same'))
  model.add(layers.LeakyReLU())
  model.add(layers.Dropout(0.3))
  model.add(layers.Flatten())
  model.add(layers.Dense(1))
  return model
```

- Discriminators classify images as fake or real:

```
discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)
```

- Define the loss function and discriminator and generator loss:

```
# This method returns a helper function to compute cross
entropy loss cross_entropy =
tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

  This method quantifies how well the discriminator is able to distinguish real images from fakes. It compares the discriminator's predictions on real images to an array of **1**s, and the discriminator's predictions on fake (generated) images to an array of **0**s:

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output),
    real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output),
    fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

- The discriminator and generator optimizers are different since we train the two networks separately:

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output),
    fake_output)
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

  Define the training loop:

```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16
# We will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF)
seed = tf.random.normal([num_examples_to_generate,
noise_dim])
```

  The following **`train_step`** is defined as a trainable function which is compiled into a TensorFlow graph.

- The training loop begins with generator receiving a random seed as input.

- This seed is used to produce an image called **generated_image**.
- The discriminator is then used to classify real images (drawn from the training set) and fakes images (produced by the generator).
- The loss is calculated for each of these models: **gen_loss** and **disc_loss**, and the gradients (**gradients_of_generator** and **gradients_of discriminator**) are used to update the generator and discriminator: **generator.trainable_variables**, **discriminator.trainable_variables**:

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
  noise = tf.random.normal([BATCH_SIZE, noise_dim])
  with tf.GradientTape() as gen_tape, tf.GradientTape() as
  disc_tape:
   generated_images = generator(noise, training=True)
   real_output = discriminator(images, training=True)
   fake_output = discriminator(generated_images,
   training=True)
   gen_loss = generator_loss(fake_output)
   disc_loss = discriminator_loss(real_output, fake_output)
  gradients_of_generator = gen_tape.gradient(gen_loss,
 generator.trainable_variables)
  gradients_of_discriminator = disc_tape.gradient(disc_loss,
 discriminator.trainable_variables)
  generator_optimizer.apply_gradients(zip(gradients_of_gener
  ator,
 generator.trainable_variables))
  discriminator_optimizer.apply_gradients(zip(gradients_of_d
  iscriminator,
 discriminator.trainable_variables))
```

- In the following train function, train_step is called for each epoch. Save the model every 15 epochs using checkpoint and final images are shown after the final epoch:

```
def train(dataset, epochs):
 for epoch in range(epochs):
  start = time.time()
  for image_batch in dataset:
        train_step(image_batch)
        # Produce images for the GIF as we go
        display.clear_output(wait=True)
        generate_and_save_images(generator,
             epoch + 1,
             seed)
        # Save the model every 15 epochs
        if (epoch + 1) % 15 == 0:
        checkpoint.save(file_prefix = checkpoint_prefix)
        print ('Time for epoch {} is {} sec'.format(epoch + 1,
     time.time()-start))
       # Generate after the final epoch
       display.clear_output(wait=True)
       generate_and_save_images(generator,
             epochs,
             seed)
```

Complete source code of the preceding sample can be found at the following link: **https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch10/dcgan_fashion_mnist.ipynb**.

The next two sections focus on defining functions to generate and save images, as well as the actual training step.

Generate and save images

Let us define a function to generate and save intermediate images created during images with input as the epoch, model, and test_input:

```
def generate_and_save_images(model, epoch, test_input):
 # Notice `training` is set to False.
 # This is so all layers run in inference mode (batchnorm).
 predictions = model(test_input, training=False)
 fig = plt.figure(figsize=(4,4))
 for i in range(predictions.shape[0]):
   plt.subplot(4, 4, i+1)
   plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5,
   cmap='gray')
   plt.axis('off')
 plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
```

```
  plt.show()
```

The next step is to train the model.

Train the model

Call the **`train()`** method defined earlier to train the generator and discriminator simultaneously. Note that training GANs can be tricky. It is important that the generator and discriminator do not overpower each other (for example, that they train at a similar rate).

At the beginning of the training, the generated images look like random noise. As the training progresses, the generated digits will look increasingly real. After about *50 epochs*, they resemble MNIST digits. This may take about one minute/epoch with the default settings on Colab:

```
train(train_dataset, EPOCHS)
```

Training output after *50 epochs* is shown in the following *figure 10.16*:

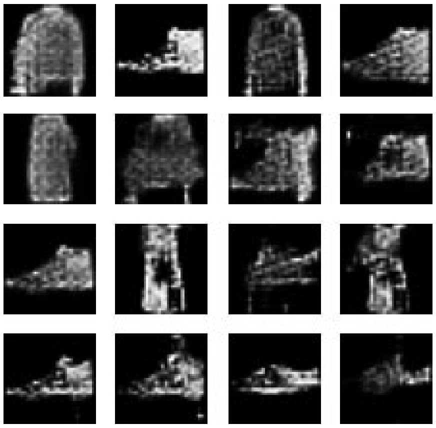


Figure 10.16: *Image generated while training*

Show the images

We will define a function to show the generated images:

```
# Display a single image using the epoch number
def display_image(epoch_no):
  return
  PIL.Image.open(‘image_at_epoch_{:04d}.png’.format(epoch_no))
display_image(EPOCHS)
anim_file = ‘dcgan.gif’
with imageio.get_writer(anim_file, mode=’I’) as writer:
  filenames = glob.glob(‘image*.png’)
  filenames = sorted(filenames)
  for filename in filenames:
    image = imageio.imread(filename)
    writer.append_data(image)
  image = imageio.imread(filename)
  writer.append_data(image)
```

The following grid shows the generated output at *10 epochs*:

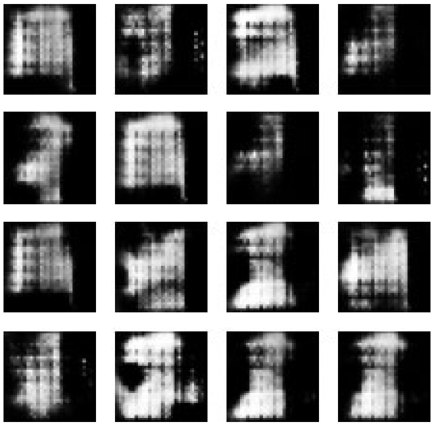


Figure 10.17: *Generated output at 10 epochs*

Notice how the features have become more prominent as compared to the image in *figure 10.17*, when the generated output is at *50 epochs*:

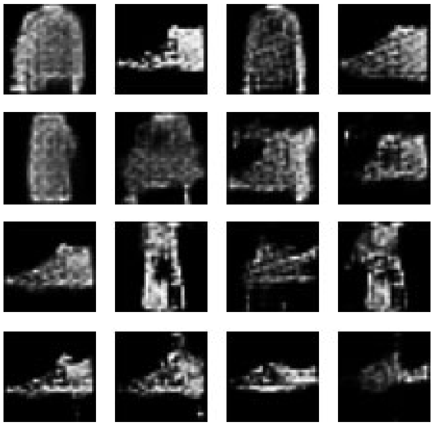


Figure 10.18: *Generated output at 50 epochs*

In the next few sections, we will look at the problem faced with the loss function being used in the DCGAN based network.

Problem with the loss function

We saw earlier the Binary Cross Entropy function, where the first part refers to the cost associated with the generator while the second part refers to the cost associated with the discriminator.

Equation for Binary Cross Entropy function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$$

Higher the value of $J(\theta)$, worse is the discriminator's performance. Equation in the *figure 10.19* shows two parts of the cost function which are affected by what the discriminator wants to minimize and what the generator wants to maximize:

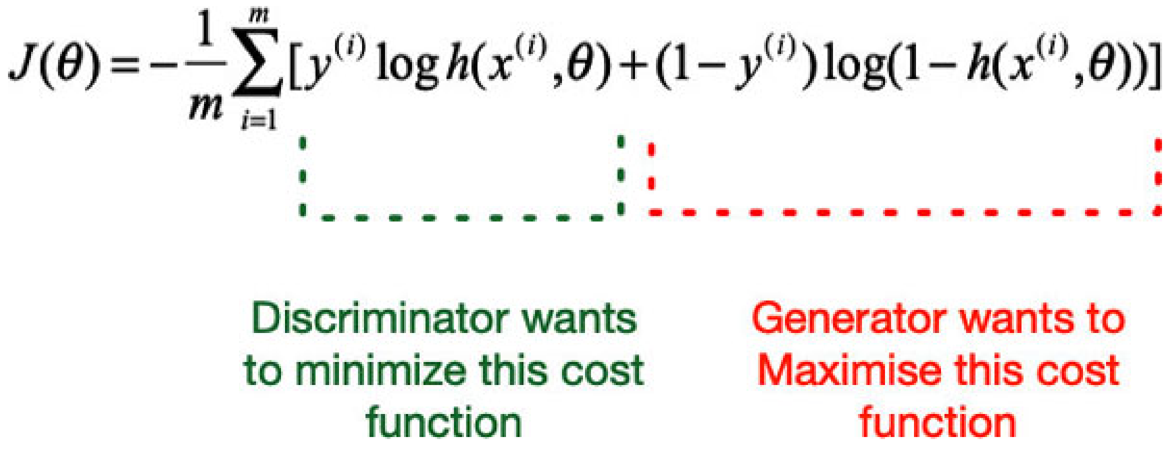

$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))]$
Discriminator wants to minimize this cost function
Generator wants to Maximise this cost function

Figure: 10.19: *Explaining two parts of the BCE loss function*

The generator wants to maximize this cost so that it can fool the discriminator to think that the fake images are real.

Since the generator has a more difficult task of generating an actual image because of BCE, after some time the cost function reaches a flat gradient as shown in the *figure 10.20*.

The goal of a GAN model is to get feature distribution of generator as close to the discriminator as possible:

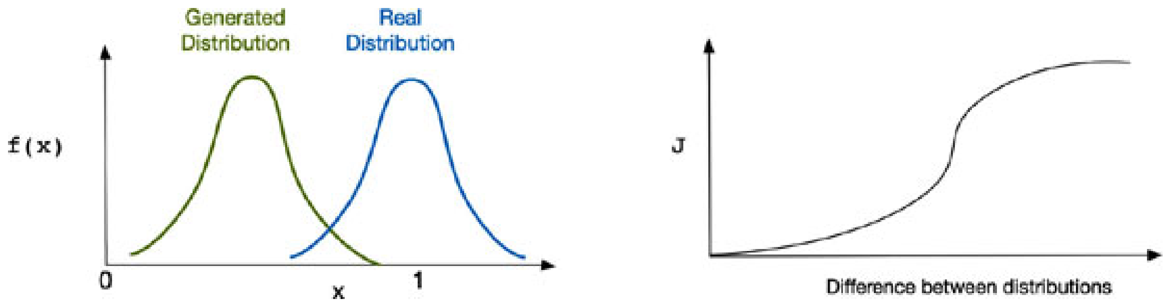


Figure 10.20: *Difference in generated versus real distribution*

Distributions in *figure 10.20* show that the loss function’s gradient vanishes as two distributions move closer. As can be seen in *figure 10.21*, the gradient of the cost function tends to zero after a few iterations:

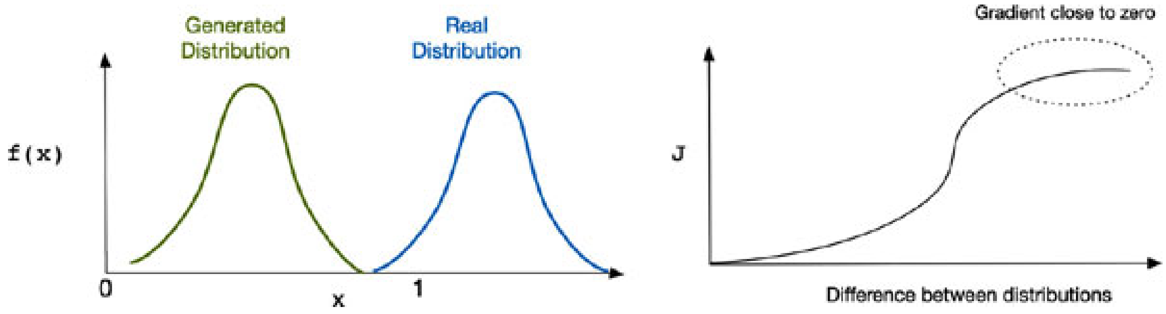


Figure 10.21: *Vanishing gradient as generator is not able to improve beyond certain epochs*

With training, the generated distribution is centered around 0, and real distribution is centered around 1.

As the discriminator improves during training and improves more quickly than the generator, the underlying cost function will have flat regions when the distributions are very different from one another. The discriminator will be able to distinguish between the real and the fake ones more easily. All of this will result in vanishing gradient problems.

Earth mover’s distance

In traditional GAN, the generator ‘s probability distribution tends to move towards 0 and that of discriminator tends to move towards 1. This distance between the two dist ributions is referred to as Earth mover’s distance. *Figure 10.22* shows the difference between these two distributions:

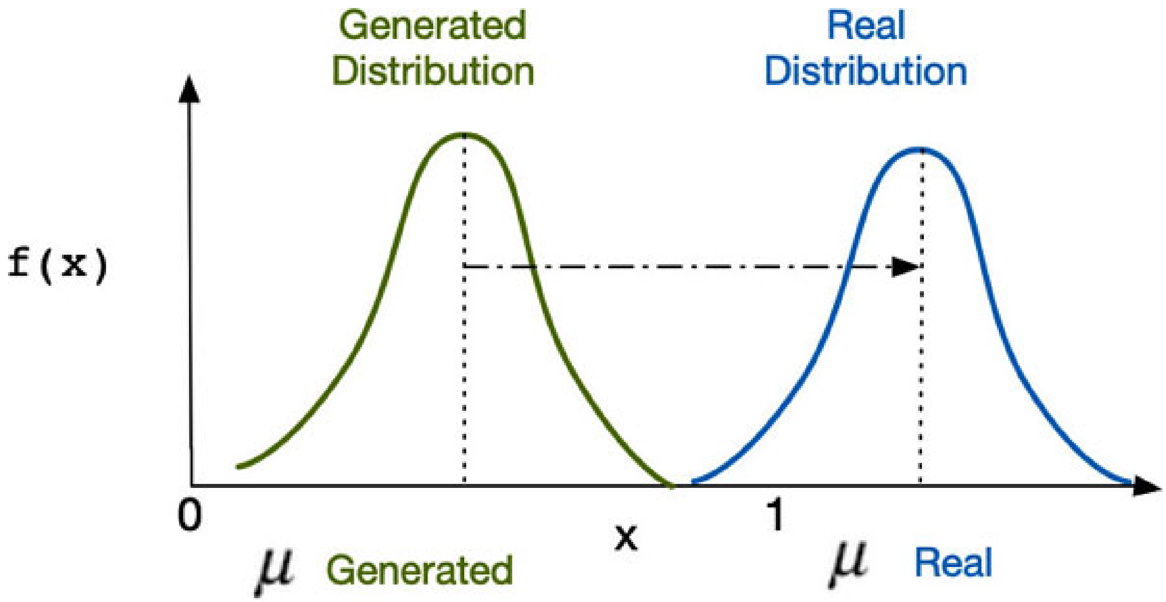


Figure 10.22: *Difference between generated and real distribution*

An effort is made to make the generated distribution as close to the actual distribution. With Earth mover's distance, there is no ceiling on how close the generated values can get to the real values, so that the cost function continues to grow. *Figure 10.23* shows the difference in distributions (real versus generated) as a function of loss function *J*:

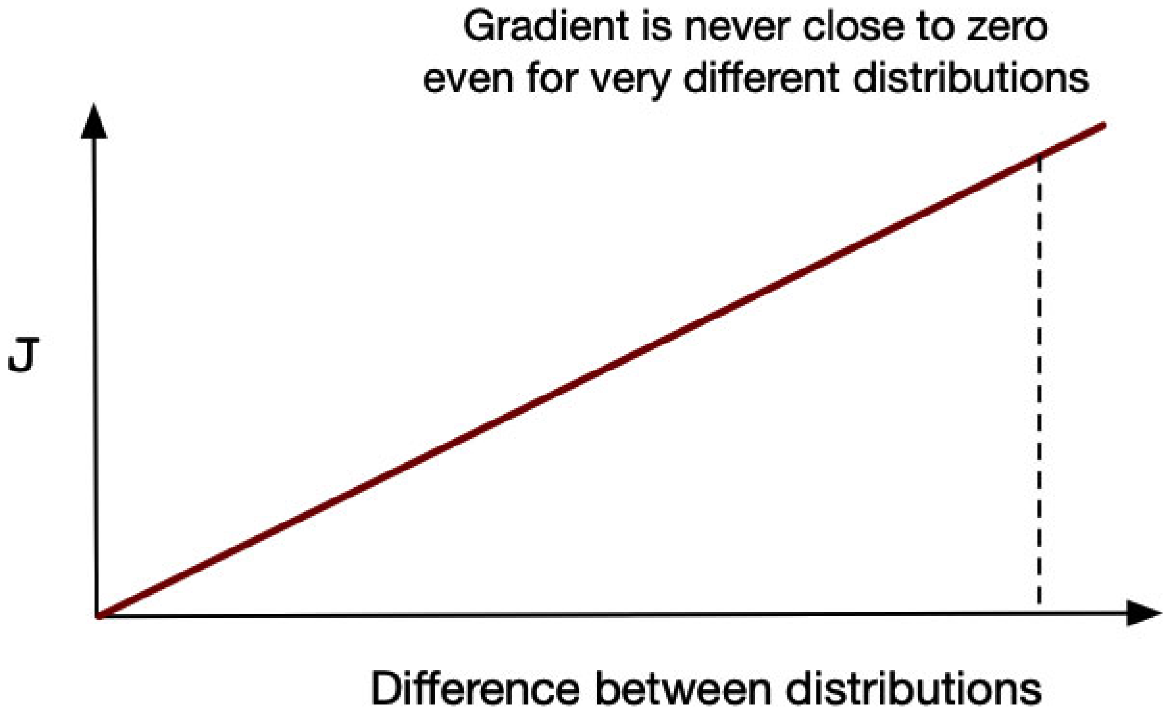


Figure: 10.23: *Difference between generator and discriminator distributions by using Earth mover's distance as a loss function.*

WGAN

This is an extension of GAN which provides an alternate way for training the generator to help with better training dataset approximation.

WGAN replaces the discriminator with a critic, which scores if image is real or fake. It is motivated by the fact that the training of generator should lead to minimization of distance between distribution of observed and training dataset. Papers talk about different distance measure distributions like Kullback-Leibler (KL) divergence, Jensen-Shannon (JS) divergence, and the Earthmover (EM) distance.

These distance measure distributions lead to impact on convergence.

Papers demonstrate that critic can be trained to approximate the *Wasserstein distance* which leads to effective generator model. Wasserstein distance provides linear gradient even after a critic is well trained.

This is quite different from the discriminator model which might fail to provide useful gradient information. Discriminator learns very quickly how to differentiate real versus fake, and does not provide reliable gradient information beyond a point.

There is a direct correlation between lower loss of critical and higher quality of generated images.

Implementing WGAN

Let us look at the changes required in DCGAN to make it a WGAN. The changes are as follows:

- linear activation function in the output layer of critic instead of sigmoid
- use of *Wasserstein loss* to train
- constrain critic model weights to a limited range after every mini batch update
- update critic more often than generator
- use of **RMSProp** version of gradient descent with smaller learning rate and no momentum

Loss function

Loss function can be summarized as:

Critic Loss = [avg critic score on real images] – [avg critic score on fake images]

Generator Loss = -[avg critic score on fake images]

Where, *avg*: **average**.

- In the case of **critic**, larger score from critic means a lower loss, and lower score from critic means a larger loss.
- In the case of **generator**, larger score on fake images is lower loss, and lower score on fake images is higher loss.

Code reference: **https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch10/WGAN_mnist_generator_critic.ipynb**.

The following class clips weights between constraints specified:

```
class ClipConstraint(Constraint):
 # set clip value when initialized
 def __init__(self, clip_value):
  self.clip_value = clip_value
 # clip model weights to hypercube
 def __call__(self, weights):
  return backend.clip(weights, -self.clip_value,
  self.clip_value)
 # get the config
 def get_config(self):
  return {'clip_value': self.clip_value}
```

Implement the *Wasserstein loss* using the following function:

```
def wasserstein_loss(y_true, y_pred):
 return backend.mean(y_true * y_pred)
```

Critic implementation:

```
def make_critic(in_shape=(28,28,1)):
 # weight initialization
 init = RandomNormal(stddev=0.02)
 # weight constraint
 const = ClipConstraint(0.01)
 # define model
 model = Sequential()
 # downsample to 14x14
 model.add(Conv2D(64, (4,4), strides=(2,2), padding='same',
 kernel_initializer=init, kernel_constraint=const,
 input_shape=in_shape))
 model.add(BatchNormalization())
 model.add(LeakyReLU(alpha=0.2))
 # downsample to 7x7
 model.add(Conv2D(64, (4,4), strides=(2,2), padding='same',
 kernel_initializer=init, kernel_constraint=const))
 model.add(BatchNormalization())
 model.add(LeakyReLU(alpha=0.2))
 # scoring, linear activation
 model.add(Flatten())
 model.add(Dense(1))
 # compile model
 opt = RMSprop(lr=0.00005)
 model.compile(loss=wasserstein_loss, optimizer=opt)
 model.summary()
 return model
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_11 (Conv2D) (None, 14, 14, 64) 1088
_________________________________________________________________
batch_normalization_14 (None, 14, 14, 64) 256
_________________________________________________________________
leaky_re_lu_17 (LeakyReLU) (None, 14, 14, 64) 0
_________________________________________________________________
conv2d_12 (Conv2D) (None, 7, 7, 64) 65600
_________________________________________________________________
batch_normalization_15 (None, 7, 7, 64) 256
_________________________________________________________________
leaky_re_lu_18 (LeakyReLU) (None, 7, 7, 64) 0
_________________________________________________________________
flatten_4 (Flatten) (None, 3136) 0
_________________________________________________________________
```

```
dense_7 (Dense) (None, 1) 3137
=================================================================
Total params: 70,337
Trainable params: 70,081
Non-trainable params: 256
```

Notice the **RMSProp** optimizer, with a learning rate of `0.00005`. Generator implementation is defined with the following method:

```
def make_generator(latent_dim):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # define model
  model = Sequential()
  # foundation for 7x7 image
  n_nodes = 128 * 7 * 7
  model.add(Dense(n_nodes, kernel_initializer=init,
  input_dim=latent_dim))
  model.add(LeakyReLU(alpha=0.2))
  model.add(Reshape((7, 7, 128)))
  # upsample to 14x14
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2),
  padding='same',
  kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # upsample to 28x28
  model.add(Conv2DTranspose(128, (4,4), strides=(2,2),
  padding='same',
  kernel_initializer=init))
  model.add(BatchNormalization())
  model.add(LeakyReLU(alpha=0.2))
  # output 28x28x1
  model.add(Conv2D(1, (7,7), activation='tanh', padding='same',
  kernel_initializer=init))
  model.summary()
  return model
```

This model has the first layer as dense, which is followed by `LeakyReLU`. It is reshaped, and `Conv2DTranspose`, batch norm, and LeakyReLU are applied for up sampling twice followed by `Conv2D` in the end.

Summary of the model is described as follows:

```
Model: "sequential_11"
_________________________________________________________________
Layer (type) Output Shape Param #
_________________________________________________________________
dense_8 (Dense) (None, 6272) 319872
_________________________________________________________________
leaky_re_lu_19 (LeakyReLU) (None, 6272) 0
_________________________________________________________________
reshape_3 (Reshape) (None, 7, 7, 128) 0
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 14, 14, 128) 262272
_________________________________________________________________
batch_normalization_16 (None, 14, 14, 128) 512
_________________________________________________________________
leaky_re_lu_20 (LeakyReLU) (None, 14, 14, 128) 0
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 28, 28, 128) 262272
_________________________________________________________________
batch_normalization_17 (None, 28, 28, 128) 512
_________________________________________________________________
leaky_re_lu_21 (LeakyReLU) (None, 28, 28, 128) 0
_________________________________________________________________
conv2d_13 (Conv2D) (None, 28, 28, 1) 6273
_________________________________________________________________
Total params: 851,713
Trainable params: 851,201
Non-trainable params: 512
_________________________________________________________________
(6265, 28, 28, 1)
```

Next, we will create the end-to-end GAN network.

Make the GAN network

1. Create a GAN network with **`generator`**, **`critic`**, and **`optimizer`** as RMSprop. Loss used is **`wasserstein_loss`**:

```
def make_gan(generator, critic):
    # make weights in the critic not trainable
    critic.trainable = False
    # connect them
    model = Sequential()
    # add generator
    model.add(generator)
    # add the critic
    model.add(critic)
    # compile model
    opt = RMSprop(lr=0.00005)
    model.compile(loss=wasserstein_loss, optimizer=opt)
    return model
```

2. Load real samples. We will load the samples from **`tensorflow.keras.datasets.mnist`** and normalize before returning (scale from **`[0,255]`** to **`[-1,1]`**):

```
def load_real_samples():
    # load dataset
    (trainX, trainy), (_, _) = load_data()
    # select all of the examples for a given class
    selected_ix = trainy == 7
    X = trainX[selected_ix]
    # expand to 3d, e.g. add channels
    X = expand dims(X, axis=-1)
    # convert from ints to floats
    X = X.astype('float32')
    # scale from [0,255] to [-1,1]
    X = (X - 127.5) / 127.5
    return X
```

3. Next, we define the function to generate real samples from the dataset given in the number of samples:

```
def generate_real_samples(dataset, n_samples):
    # choose random instances
    ix = randint(0, dataset.shape[0], n_samples)
    # select images
    X = dataset[ix]
    # generate class labels, -1 for 'real'
    y = -ones((n_samples, 1))
    return X, y
```

4. Define the function to generate points in latent space as input for the generator:

```
def generate_latent_points(latent_dim, n_samples):
    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    # reshape into a batch of inputs for the network
    x_input = x_input.reshape(n_samples, latent_dim)
    return x_input
```

5. Use the generator to generate **n** fake examples with class labels:

```
def generate_fake_samples(generator, latent_dim, n_samples):
    # generate points in latent space
    x_input = generate_latent_points(latent_dim, n_samples)
    # predict outputs
    X = generator.predict(x_input)
    # create class labels with 1.0 for 'fake'
    y = ones((n_samples, 1))
    return X, y
```

6. Generate samples and save as a plot, and save the model:

```
def summarize_performance(step, g_model, latent_dim,
n_samples=100):
    # prepare fake examples
    X, _ = generate_fake_samples(g_model, latent_dim,
    n_samples)
    # scale from [-1,1] to [0,1]
    X = (X + 1) / 2.0
    # plot images
    for i in range(10 * 10):
     # define subplot
     pyplot.subplot(10, 10, 1 + i)
     # turn off axis
     pyplot.axis('off')
     # plot raw pixel data
     pyplot.imshow(X[i, :, :, 0], cmap='gray_r')
    # save plot to file
    filename1 = 'generated_plot_%04d.png' % (step+1)
    pyplot.savefig(filename1)
    pyplot.close()
    # save the generator model
```

```
  filename2 = 'model_%04d.h5' % (step+1)
  g_model.save(filename2)
  print('>Saved: %s and %s' % (filename1, filename2))
```

7. Define the `train` function:

```
from datetime import datetime
now = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
print(now)
filename = 'wgan_losses_' + now + '.csv'
print(filename)
file = open(filename, 'w+', newline ='\n')
def train(g_model, c_model, gan_model, dataset, latent_dim,
n_epochs=1,
n_batch=64, n_critic=5):
  # calculate the number of batches per training epoch
  bat_per_epo = int(dataset.shape[0] / n_batch)
  # calculate the nufile = open('g4g.csv', 'w+', newline
  ='') mber of training iterations
  n_steps = bat_per_epo * n_epochs
  # calculate the size of half a batch of samples
  half_batch = int(n_batch / 2)
  # lists for keeping track of loss
  c1_hist, c2_hist, g_hist = list(), list(), list()
  # manually enumerate epochs
  for i in range(n_steps):
   # update the critic more than the generator
   c1_tmp, c2_tmp = list(), list()
   for _ in range(n_critic):
    # get randomly selected 'real' samples
    X_real, y_real = generate_real_samples(dataset,
    half_batch)
    # update critic model weights
    c_loss1 = c_model.train_on_batch(X_real, y_real)
    c1_tmp.append(c_loss1)
    # generate 'fake' examples
    X_fake, y_fake = generate_fake_samples(g_model,
    latent_dim,
 half_batch)
    # update critic model weights
```

```
      c_loss2 = c_model.train_on_batch(X_fake, y_fake)
      c2_tmp.append(c_loss2)
     # store critic loss
     c1_hist.append(mean(c1_tmp))
     c2_hist.append(mean(c2_tmp))
     # prepare points in latent space as input for the
     generator
     X_gan = generate_latent_points(latent_dim, n_batch)
     # create inverted labels for the fake samples
     y_gan = -ones((n_batch, 1))
     # update the generator via the critic's error
     g_loss = gan_model.train_on_batch(X_gan, y_gan)
     g_hist.append(g_loss)
     # summarize loss on this batch
     print('>%d, c1=%.3f, c2=%.3f g=%.3f' % (i+1, c1_hist[-1],
     c2_hist[-1], g_loss))
     file.write(str(i+1) + ", " + str(c1_hist[-1]) + ", " +
     str(c2_hist[-1]) + ", " + str(g_loss) + "\n" )
     file.flush()
     # evaluate the model performance every 'epoch'
     if (i+1) % bat_per_epo == 0:
      summarize_performance(i, g_model, latent_dim)
    return c1_hist, c2_hist, g_hist
```

8. Call the **`train`** function:

```
train(generator, critic, gan_model, dataset, latent_dim)
```

This will output the performance for each batch and plot the chart. The output will print losses for each iteration and save the model finally:

```
print('>%d, c1=%.3f, c2=%.3f g=%.3f' % (i+1, c1_hist[-1],
c2_hist[-1], g_loss))
>1, c1=-1.713, c2=-0.043 g=0.002
>2, c1=-6.165, c2=0.003 g=-0.008
>3, c1=-9.015, c2=0.027 g=-0.015
...
>95, c1=-71.196, c2=-64.475 g=-123.603
>96, c1=-70.998, c2=-65.188 g=-127.017
>97, c1=-72.178, c2=-66.080 g=-129.777
>Saved: generated_plot_0097.png and model_0097.h5
```

9. Lastly, we would plot these losses:

```
plot_history(c1_hist, c2_hist, g_hist)
```

The following *figure 10.24* will be plotted as the output. It shows how the generator loss keeps on going down:

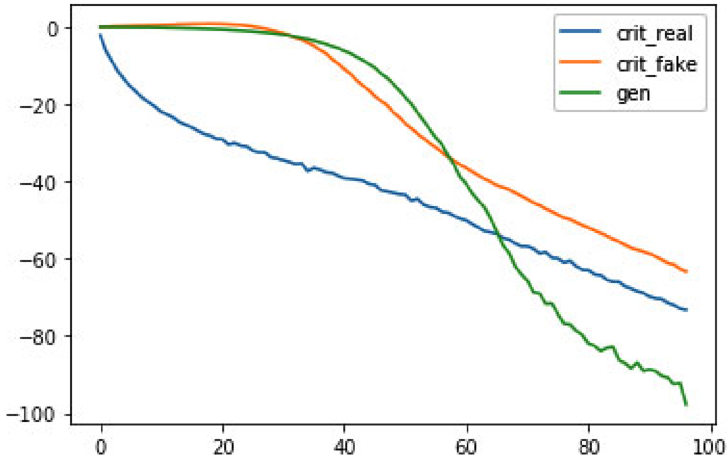


Figure 10.24: *Losses for WGAN generator and critic on real and fake images*

Let us look at the variation of WGAN. If we don't clip the gradients, the generator loss comes down almost the same, but critic losses taper off.

If we apply ReLU, instead of WGAN, notice how the losses don't come down by the same level as shown in the *figure 10.25*:

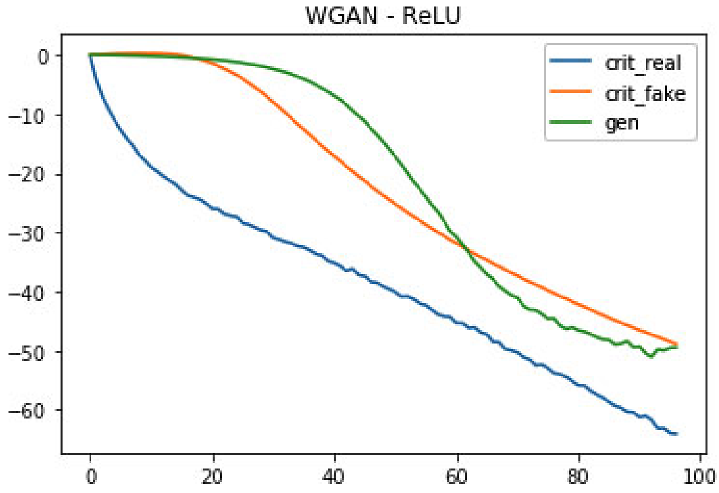


Figure 10.25: *Losses for WGAN generator and critic on real and fake images with ReLU loss function for generator*

```
Figure-Caption--BPB-HEB para-style-override-112
```

If we don't clip the gradients, let us look at how the graph changes, as shown in *figure 10.26*:

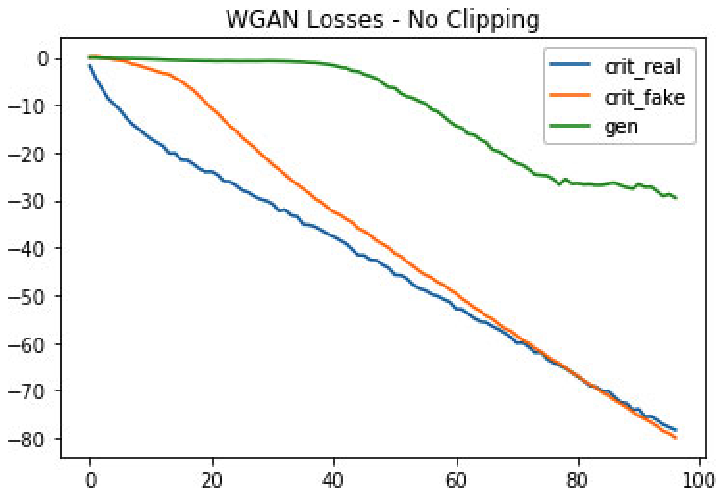


Figure 10.26: *Losses for WGAN generator and critic on real and fake images with no clipping*

Notice how the generator losses taper off and critic losses keep going down. With this, we come to the end of this section.

Pix2Pix with Maps dataset

This section and the sample will show how to build and train a **conditional generative adversarial network** (**cGAN**) also known as **pix2pix**. It learns mapping from input images to output images, as described in image-to-image translation with conditional adversarial networks (**https://arxiv.org/abs/1611.07004**) by *Isola et al. (2017)*. Pix2pix is not application specific. It can be applied to a wide range of tasks, including synthesizing photos from label maps, getting colorized photos from black and white images, turning Google Maps photos into aerial images, and so on. It can even transform sketches into photos. In this example, your network will generate images of building maps using the **Maps** database provided by the *Electrical Engineering department at University of Berkley (***https://arxiv.org/abs/1611.07004***)*. In the **pix2pix cGAN**, we condition on input images and generate corresponding output images. cGANs were first proposed in *Conditional Generative Adversarial Nets (***https://arxiv.org/abs/1411.1784***)* (*Mirza* and *Osindero*, *2014*). The architecture of this network will contain:

- a generator with a **U-Net-based architecture**.
- a discriminator represented by a convolutional **PatchGAN** classifier (proposed in the *pix2pix paper* (**https://arxiv.org/abs/1611.07004**). Each epoch can take around *15 seconds* depending on the GPU used. The

following *figure 10.27* shows some examples of the output generated by the *pix2pix cGAN* after training for *200 epochs* on the **Maps** dataset (4K steps):

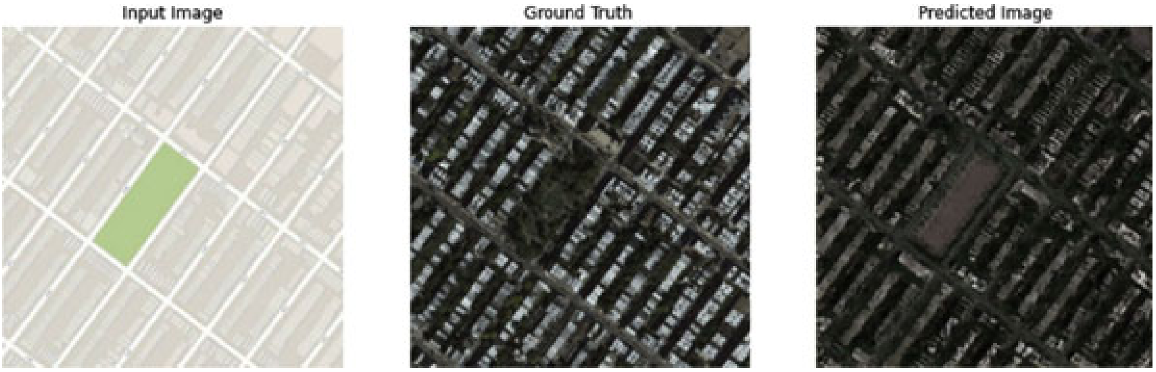


Figure 10.27: *Example of input image, ground truth, and predicted image*

Let us process the dataset by importing the relevant packages from tensorflow, os, time, matplotlib, and so on:

1. Import TensorFlow libraries:

```
import tensorflow as tf
import os
import pathlib
import time
import datetime
from matplotlib import pyplot as plt
from IPython import display
```

2. Next, load the dataset:

```
dataset_name = "maps"
_URL =
f'http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/{dataset_name}.tar.gz'
path_to_zip = tf.keras.utils.get_file (
   fname=f"{dataset_name}.tar.gz",
   origin=_URL,
   extract=True)
path_to_zip = pathlib.Path(path_to_zip)
PATH = path_to_zip.parent/dataset_name
list(PATH.parent.iterdir())
```

3. Each original image is of size *600 x 1200* containing two *600 x 600* images:

```
sample_image = tf.io.read_file(str(PATH / 'train/1.jpg'))
sample_image = tf.io.decode_jpeg(sample_image)
print(sample_image.shape)
sample_image_2 = tf.io.read_file(str(PATH / 'train/2.jpg'))
sample_image_2 = tf.io.decode_jpeg(sample_image_2)
plt.figure()
plt.imshow(sample_image)
```

The output of the `imshow(..)` command is shown in *figure 10.28*:

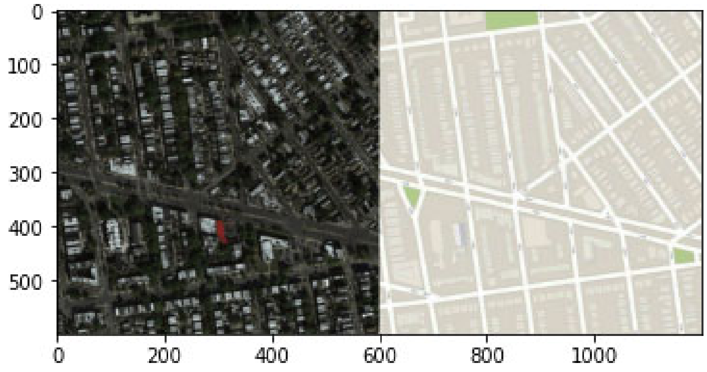


***Figure 10.28:** Output of dataset image 1.jpg*

Let us look at another image, as shown in *figure 10.29*:

```
plt.figure()
plt.imshow(sample_image_2)
```

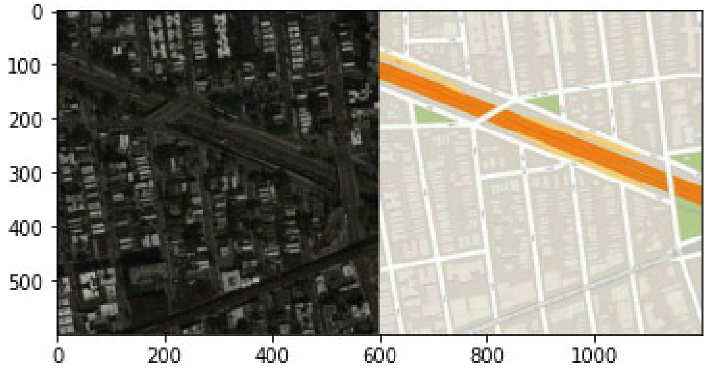


***Figure 10.29:** Output of dataset image 2.jpg*

4. We will need to separate real map images from the annotated map images—all of which will be of size *600 x 600*.
5. Let us define a function that loads image files and outputs two image tensors:

```
def load(image_file):
    # Read and decode an image file to a uint8 tensor
    image = tf.io.read_file(image_file)
    image = tf.image.decode_jpeg(image)
    # Split each image tensor into two tensors:
    # - one with a real building facade image
    # - one with an architecture label image
    w = tf.shape(image)[1]
    w = w // 2
    input_image = image[:, w:, :]
    real_image = image[:, :w, :]
    # Convert both images to float32 tensors
    input_image = tf.cast(input_image, tf.float32)
    real_image = tf.cast(real_image, tf.float32)
    return input_image, real_image
```

6. Plot a sample of the input (architecture label image) and real (map) images:

```
inp, re = load(str(PATH / 'train/100.jpg'))
# Casting to int for matplotlib to display the images
plt.figure()
plt.imshow(inp / 255.0)
plt.figure()
plt.imshow(re / 255.0)
```

Figure 10.30 shows the architecture labeled image and real image.

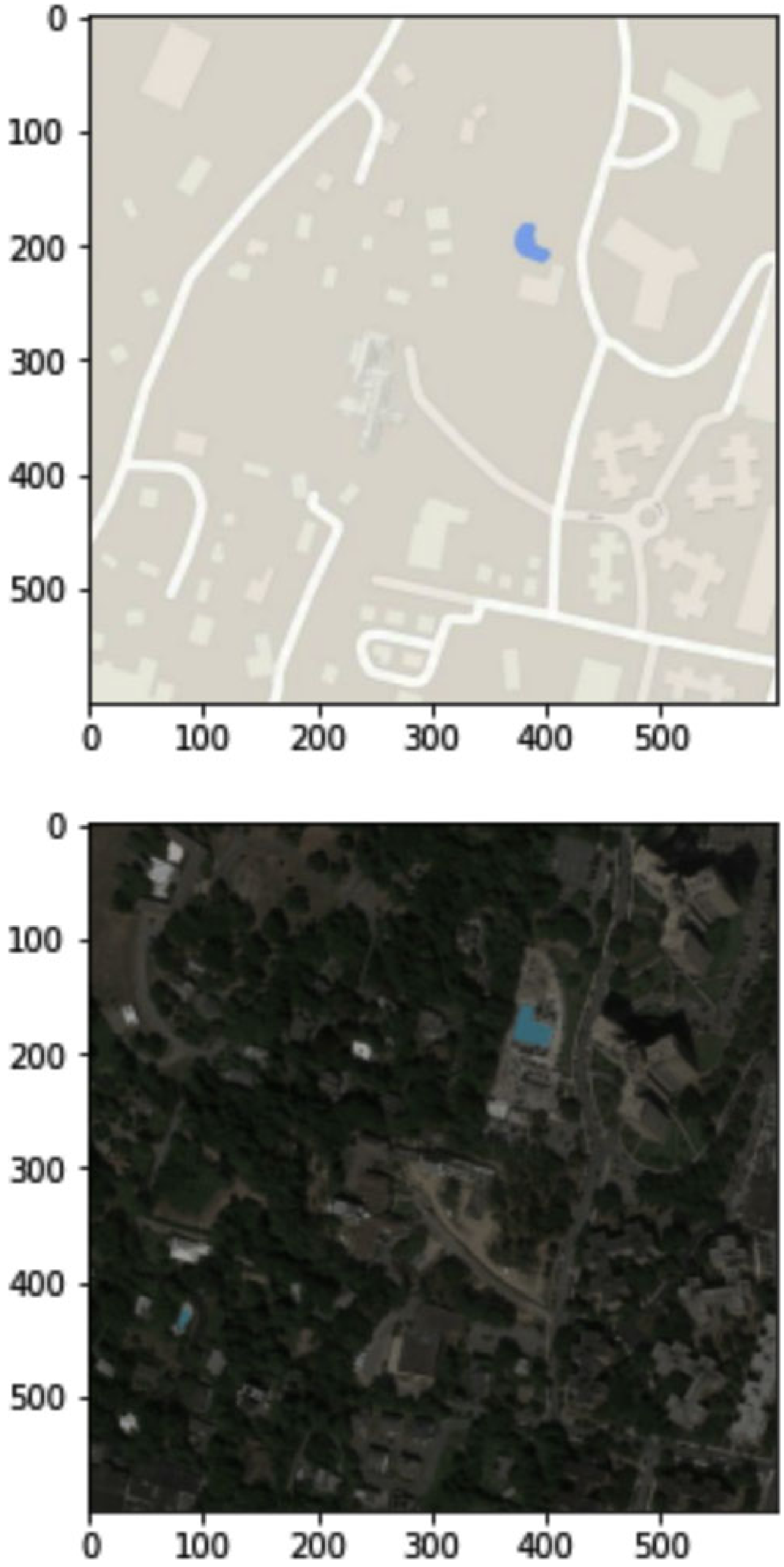


Figure 10.30: *Output of sample and real image after splitting*

As described in the *pix2pix paper* (**https://arxiv.org/abs/1611.07004**), we will apply random jittering and mirroring to preprocess the training set:

Define several functions that:

- resize each *600 x 600* image to a larger height and width—*800 x 800.*
- randomly crop it back to *600 x 600.*
- randomly flip the image horizontally, that is, left to right (random mirroring).
- normalize the images to the `[-1, 1]` range:

```
# The map training set consist of x images
BUFFER_SIZE = 400
# The batch size of 1 produced better results for the U-
Net in the original pix2pix experiment
BATCH_SIZE = 1
# Each image is 600x600 in size
IMG_WIDTH = 600
IMG_HEIGHT = 600
```

7. Resize function as follows:

```
def resize(input_image, real_image, height, width):
  input_image = tf.image.resize(input_image, [height, width],
        method=tf.image.ResizeMethod.NEAREST_ NEIGHBOR)
  real_image = tf.image.resize(real_image, [height, width],
        method=tf.image.ResizeMethod.NEAREST_ NEIGHBOR)
  return input_image, real_image
```

8. Randomly crop as follows:

```
def random_crop(input_image, real_image):
     stacked_image = tf.stack([input_image, real_image],
     axis=0)
     cropped_image = tf.image.random_crop(
    stacked_image, size=[2, IMG_HEIGHT, IMG_WIDTH, 3])
     return cropped_image[0], cropped_image[1]
```

9. Normalize the images between -1 and 1:

```
def normalize(input_image, real_image):
     input_image = (input_image / 127.5) - 1
     real_image = (real_image / 127.5) - 1
```

```
return input_image, real_image
```

10. Define a function to apply random jitter:

```
@tf.function()
def random_jitter(input_image, real_image):
# Resizing to 800x800
input_image, real_image = resize(input_image, real_image,
800, 800)
# Random cropping back to 600x600
input_image, real_image = random_crop(input_image,
real_image)
if tf.random.uniform(()) > 0.5:
  # Random mirroring
  input_image = tf.image.flip_left_right(input_image)
  real_image = tf.image.flip_left_right(real_image)
return input_image, real_image
```

All the preceding steps help improve the size of the dataset. Let us plot the output of the random jitter function:

```
plt.figure(figsize=(6, 6))
for I in range(4):
     rj_inp, rj_re = random_jitter(inp, re)
     plt.subplot(2, 2, i + 1)
     plt.imshow(rj_inp / 255.0)
     plt.axis`'of'')
plt.show()
```

The plot output is shown in *figure 10.31*:

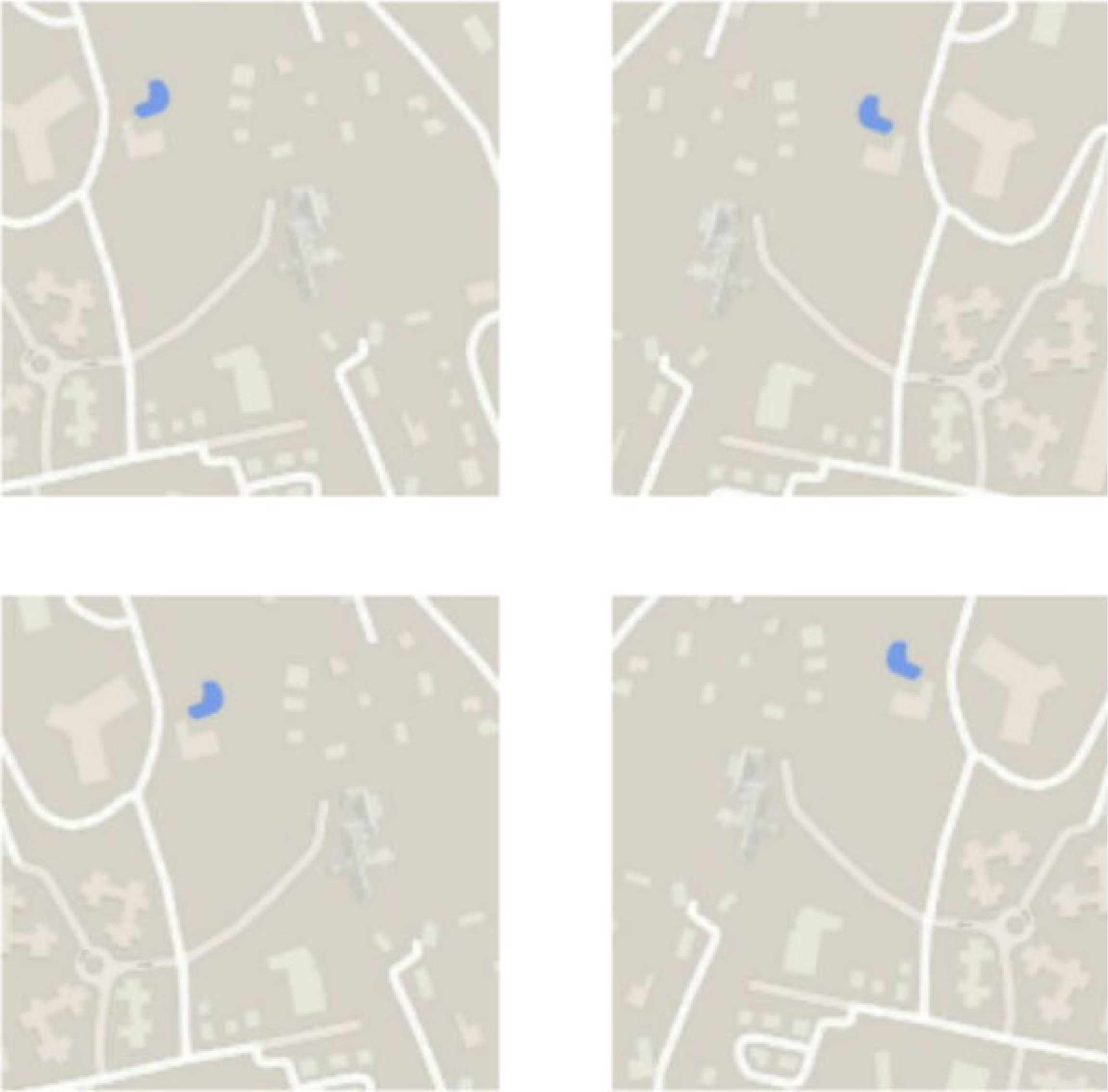


Figure 10.31: *Output of sample image after rotating and cropping*

Having checked that the loading and preprocessing works, let’s define a couple of helper functions that load and preprocess the training and test sets:

```
def load_image_train(image_file):
  input_image, real_image = load(image_file)
  input_image, real_u
```

Build the generator

The generator of the pix2pix cGAN is a modified U-Net (**https://arxiv.org/abs/1505.04597**). A U-Net consists of an **encoder** (down sampler) and **decoder** (up sampler). (Please find more about it in the image segmentation tutorial in the following link:

https://www.tensorflow.org/tutorials/images/segmentation

and on the **U-Net project website** in the following link:

https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/)

- Each block in the encoder is: **Convolution** -> **Batch normalization** -> **Leaky ReLU**.
- Each block in the decoder is: **Transposed convolution** -> **Batch normalization** -> **Dropout** (applied to the first three blocks) -> **ReLU**.

There are skip connections between the encoder and the decoder (as in the U-Net):

```
OUTPUT_CHANNELS = 3
def downsample(filters, size, apply_batchnorm=True):
 initializer = tf.random_normal_initializer(0., 0.02)
 result = tf.keras.Sequential()
 result.add(
  tf.keras.layers.Conv2D(filters, size, strides=2,
  padding='same', kernel_initializer=initializer,
        use_bias=False))
 if apply_batchnorm:
  result.add(tf.keras.layers.BatchNormalization())
 result.add(tf.keras.layers.LeakyReLU())
 return result
down_model = downsample(3, 4)
down_result = down_model(tf.expand_dims(inp, 0))
print (down_result.shape)
```

Output

```
(1, 300, 300, 3)
```

Next, let us look at the up sampler

```
def upsample(filters, size, apply_dropout=False):
 initializer = tf.random_normal_initializer(0., 0.02)
 result = tf.keras.Sequential()
 result.add(
 tf.keras.layers.Conv2DTranspose(filters, size, strides=2,
         padding='same',
          kernel_initializer=initializer,
          use_bias=False))
  result.add(tf.keras.layers.BatchNormalization())
  if apply_dropout:
  result.add(tf.keras.layers.Dropout(0.5))
  result.add(tf.keras.layers.ReLU())
  return result
up_model = upsample(3, 4)
up_result = up_model(down_result)
print (up_result.shape)
    (1, 600, 600, 3)
```

Define a generator where we output a model with **`up_stack`** and **`down_stack`** layers combined together:

```
def Generator():
  inputs = tf.keras.layers.Input(shape=[600, 600, 3])
  down_stack = [
    downsample(150, 4, apply_batchnorm=False),# (batch_size, 128,
    128, 64)
    downsample(300, 4, apply_batchnorm=False),# (batch_size, 128,
    128, 64)
    downsample(600, 4, apply_batchnorm=False),
  ]
  up_stack = [
    upsample(600, 4),
    upsample(300, 4),
    upsample(150, 4),
  ]
  initializer = tf.random_normal_initializer(0., 0.02)
  last = tf.keras.layers.Conv2DTranspose(OUTPUT_CHANNELS, 4,
        strides=2,
        padding='same',
        kernel_initializer=initializer,
        activation='tanh')
  # (batch_size, 600, 600, 3)
  x = inputs
  # Downsampling through the model
  skips = []
  for down in down_stack:
   x = down(x)
   skips.append(x)
  skips = reversed(skips[:-1])
  # Upsampling and establishing the skip connections
  for up, skip in zip(up_stack, skips):
   x = up(x)
   x = tf.keras.layers.Concatenate()([x, skip])
  x = last(x)
  return tf.keras.Model(inputs=inputs, outputs=x)
```

Visualize the generator as follows:

```
generator = Generator()
tf.keras.utils.plot_model(generator, show_shapes=True, dpi=64)
```

The output shown in *figure 10.32* shows how down sampling and up sampling layers are stacked next to each other:

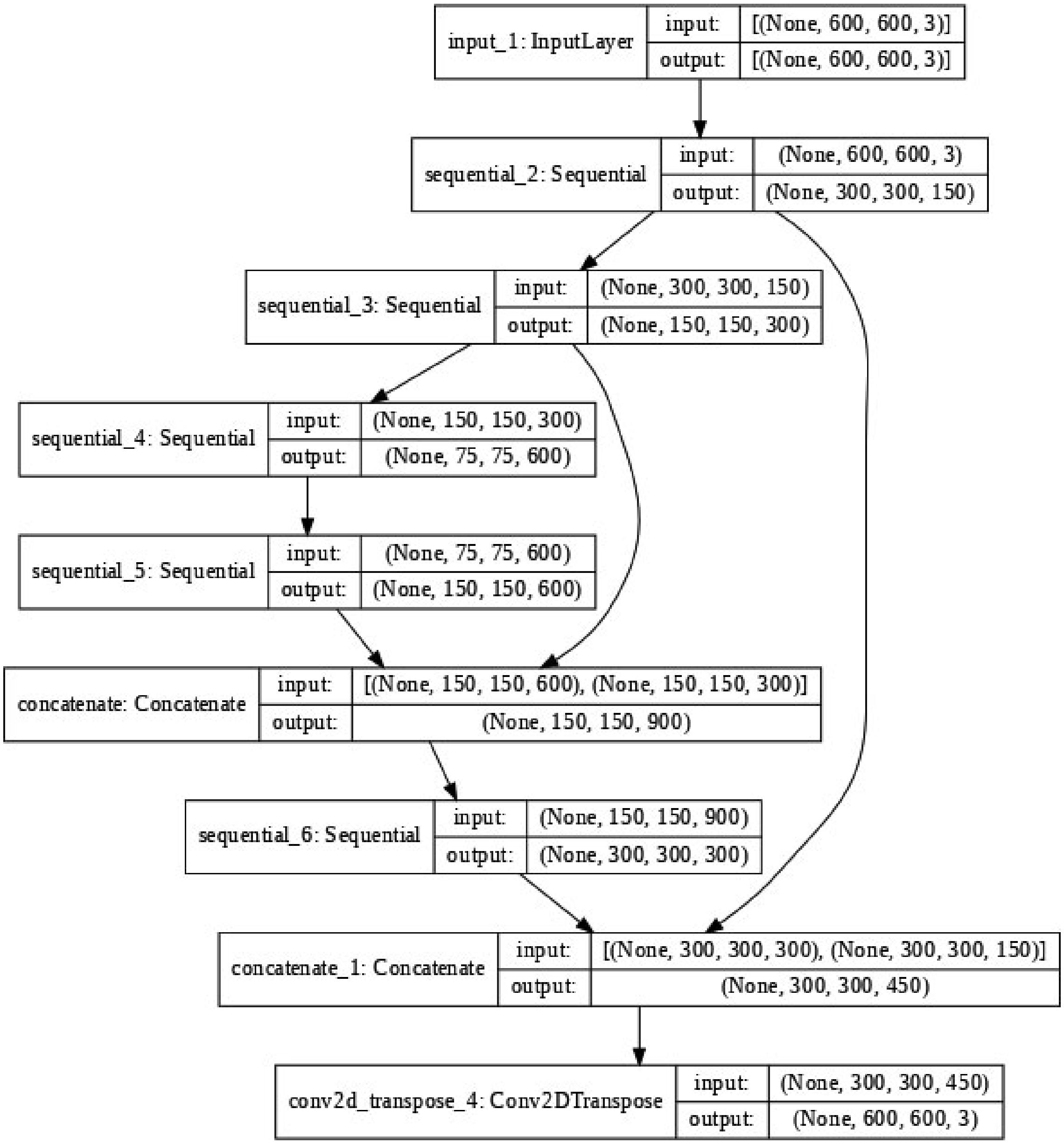


Figure 10.32: *Generator model topology*

Next, we look at the generator loss.

GANs learn a loss that adapts to the data, while cGANs learn a structured loss that penalizes a possible structure that differs from the network output and the target image, as described in the *pix2pix paper* (**https://arxiv.org/abs/1611.07004**).

The generator loss is a sigmoid cross-entropy loss of the generated images and an array of ones.

The *pix2pix paper* also mentions the **L1 loss**, which is a **mean absolute error** (**MAE**) between the generated image and the target image.

This allows the generated image to become structurally similar to the target image.

The formula to calculate the total generator loss is *gan_loss + LAMBDA * l1_loss*, where *LAMBDA = 100*. This value was decided by the authors of the paper.

```
LAMBDA = 100
loss_object =
tf.keras.losses.BinaryCrossentropy(from_logits=True)
def generator_loss(disc_generated_output, gen_output, target):
  gan_loss = loss_object(tf.ones_like(disc_generated_output),
  disc_generated_output)
  # Mean absolute error
  l1_loss = tf.reduce_mean(tf.abs(target - gen_output))
  total_gen_loss = gan_loss + (LAMBDA * l1_loss)
      return total_gen_loss, gan_loss, l1_loss
```

The training procedure for the generator is shown in *figure 10.32*.

The discriminator in the pix2pix cGAN is a convolutional PatchGAN classifier. It tries to classify if each image ***patch*** is real or not real, as described in the *pix2pix paper* (**https://arxiv.org/abs/1611.07004**).

- Each block in the discriminator is: **Convolution** -> **Batch normalization** -> **Leaky ReLU**.
- The shape of the output after the last layer is (**`batch_size`**, **`30`**, **`30`**, **`1`**).
- Each *30 x 30* image patch of the output classifies a *70 x 70* portion of the input image.
- The discriminator receives two inputs:
    - The input image and the target image, which it should classify as real.
    - The input image and the generated image (the output of the generator), which it should classify as fake.
    - Use **`tf.concat([inp, tar], axis=-1)`** to concatenate these two inputs together.

Let us look at the sequence of schematics of finding the loss and using it to update the gradients:

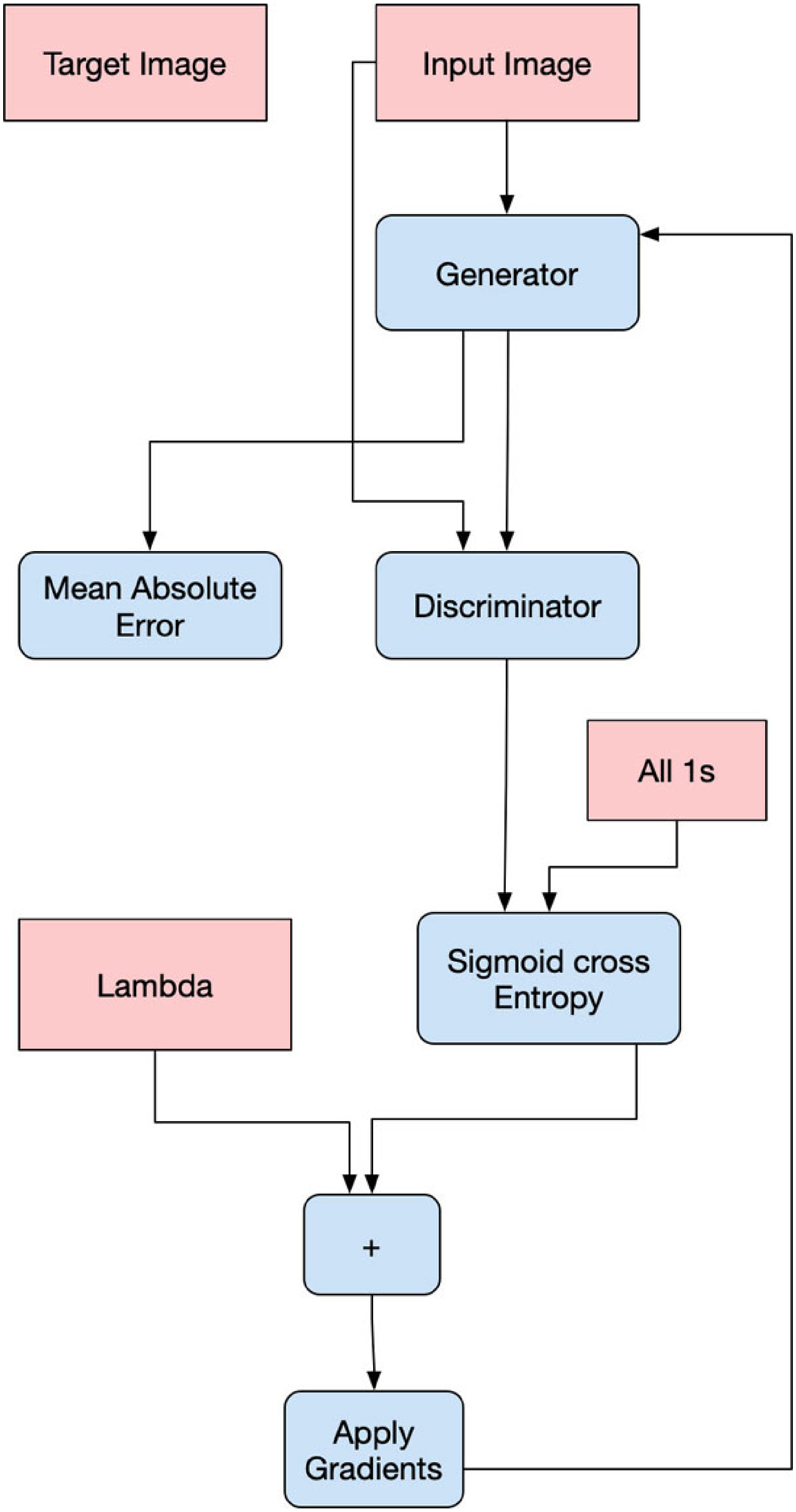

Target Image
Input Image
Generator
Mean Absolute Error
Discriminator
All 1s
Lambda
Sigmoid cross Entropy
+
Apply Gradients

Figure 10.33: *Generator loss logic*

Discriminator:

```
def Discriminator():
  initializer = tf.random_normal_initializer(0., 0.02)
  inp = tf.keras.layers.Input(shape=[600, 600, 3],
  name='input_image')
  tar = tf.keras.layers.Input(shape=[600, 600, 3],
  name='target_image')
  x = tf.keras.layers.concatenate([inp, tar]) # (batch_size, 256,
  256, channels*2)
  down1 = downsample(150, 4, False)(x) # (batch_size, 128, 128,
  64)
  down2 = downsample(300, 4)(down1) # (batch_size, 64, 64, 128)
  down3 = downsample(600, 4)(down2) # (batch_size, 32, 32, 256)
  zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3)
  # (batch_size, 34, 34, 256)
  conv = tf.keras.layers.Conv2D(512, 4, strides=1,
          kernel_initializer=initializer,
          use_bias=False)(zero_pad1)
  # (batch_size, 31, 31, 512)
  batchnorm1 = tf.keras.layers.BatchNormalization()(conv)
  leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)
  zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu)
  # (batch_size, 33, 33, 512)
  last = tf.keras.layers.Conv2D(1, 4, strides=1,
  kernel_initializer=initializer)(zero_pad2) # (batch_size, 30,
  30, 1)
  return tf.keras.Model(inputs=[inp, tar], outputs=last)
```

Let us look at the topology as displayed in *figure 10.34*:

```
discriminator = Discriminator()
tf.keras.utils.plot_model(discriminator, show_shapes=True,
dpi=64)
```

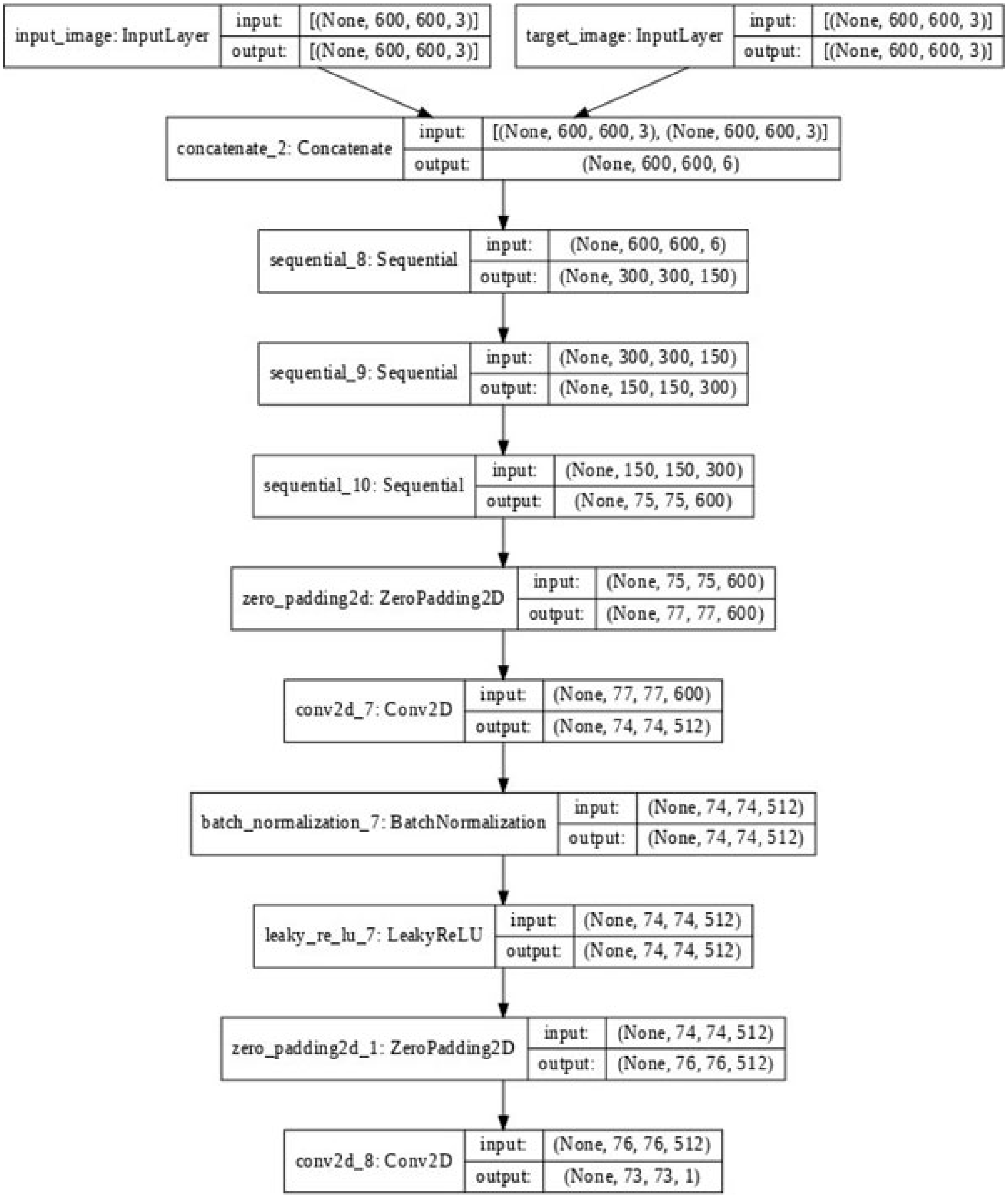


Figure 10.34: *Topology of the generator*

Next, let us define the discriminator loss:

- The **`discriminator_loss`** function takes two inputs: **real images** and **generated images**.
- **`real_loss`** is a sigmoid cross-entropy loss of the real images and an array of ones (since these are the real images).

- **`generated_loss`** is a sigmoid cross-entropy loss of the **generated images** and an array of zeros (since these are the fake images).
- The **`total_loss`** is the sum of **`real_loss`** and **`generated_loss`**:

```
def discriminator_loss(disc_real_output,
disc_generated_output):
     real_loss = loss_object(tf.ones_like(disc_real_output),
     disc_ real_output)
     generated_loss =
     loss_object(tf.zeros_like(disc_generated_ output),
  disc_generated_output)
     total_disc_loss = real_loss + generated_loss
return total_disc_loss
```

Optimizers and checkpoints

Next, we will define optimizers for the generator and the discriminator:

```
generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(2e-4,
  beta_1=0.5)
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint =
tf.train.Checkpoint(generator_optimizer=generator_optimizer,
discriminator_optimizer=
discriminator_optimizer, generator=generator,
discriminator=discriminator)
```

We will be using Adam Optimizer for both the generator and the discriminator.

Plot images during training

Let us first create a function to plot generated images during training:

- Write a function to plot some images during training.
- Pass images from the test set to the generator.
- The generator will then translate the input image into the output.
- The last step is to plot the predictions and inspect the images.

The `training=True` is intentional here since we want the batch statistics while running the model on the test dataset. If we use `training=False`,

we will get the accumulated statistics learned from the training dataset (which we don't want).

```
def generate_images(model, test_input, tar):
  prediction = model(test_input, training=True)
  plt.figure(figsize=(15, 15))
  display_list = [test_input[0], tar[0], prediction[0]]
  title = ['Input Image', 'Ground Truth', 'Predicted Image']
  for i in range(3):
    plt.subplot(1, 3, i+1)
    plt.title(title[i])
    # Getting the pixel values in the [0, 1] range to plot.
    plt.imshow(display_list[i] * 0.5 + 0.5)
    plt.axis('off')
  plt.show()
```

Let us test the function by executing it:

```
for example_input, example_target in test_dataset.take(1):
    generate_images(generator, example_input, example_target)
```

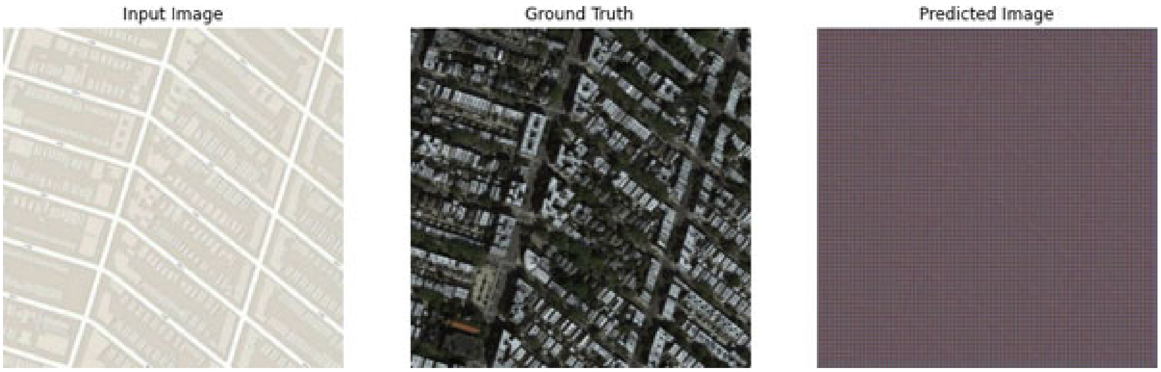


Figure 10.35: *Comparing the input image, ground truth and predicted image*

Let us put everything together by defining `tf` function for the training loop.

Training

The steps in each training step called `train_step(..)` are defined as follows:

- For each example, input generates an output.
- The discriminator receives the `input_image` and the generated image as the first input. The second input is the `input_image` and the `target_image`.

- Next, calculate the generator and discriminator loss.
- Then, calculate the gradients of loss with respect to both the generator and discriminator variables (inputs), and apply those to the optimizer.
- Finally, log the losses to TensorBoard:

```
log_dir="logs/" summary_writer =
tf.summary.create_file_writer( log_dir + "fit/" +
datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
@tf.function
def train_step(input_image, target, step):
with tf.GradientTape() as gen_tape, tf.GradientTape() as
disc_ tape:
  gen_output = generator(input_image, training=True)
  disc_real_output = discriminator([input_image, target],
  training=True)
  disc_generated_output = discriminator([input_image, gen_
  output],
  training=True)
  gen_total_loss, gen_gan_loss, gen_l1_loss =
  generator_loss(
 disc_generated_output, gen_output, target)
  disc_loss = discriminator_loss(disc_real_output, disc_
  generated_output)
generator_gradients = gen_tape.gradient(gen_total_loss,
generator.trainable_variables)
discriminator_gradients = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
generator_optimizer.apply_gradients(zip(generator_gradients,
generator.trainable_variables))
discriminator_optimizer.apply_gradients(zip(discriminator_gr
adients,
  discriminator.trainable_variables))
with summary_writer.as_default():
  tf.summary.scalar('gen_total_loss', gen_total_loss,
  step=step//1000)
  tf.summary.scalar('gen_gan_loss', gen_gan_loss,
  step=step//1000)
  tf.summary.scalar('gen_l1_loss', gen_l1_loss,
  step=step//1000)
```

```
      tf.summary.scalar('disc_loss', disc_loss, step=step//1000)
```

The actual training loop

Since these samples can run on more than one dataset, and the datasets vary greatly in size, the training loop is setup to work in steps instead of epochs.

Let us first define a function to train based on the training data: **`train_dataset`** and test it against **`test_dataset`** and run it for *n* steps:

- Iterate over the number of steps.
- At every **`10`** steps, print a dot (`.`).
- At every **`1k`** steps, clear the display and run **`generate_images`** to show the progress.
- At every **`5k`** steps, save a checkpoint:

```
def fit(train_ds, test_ds, steps):
example_input, example_target = next(iter(test_ds.take(1)))
start = time.time()
for step, (input_image, target) in
train_ds.repeat().take(steps).enumerate():
  if (step) % 1000 == 0:
  display.clear_output(wait=True)
  if step != 0:
   print(f'Time taken for 1000 steps: {time.time()-
   start:.2f} sec\n')
  start = time.time()
  generate_images(generator, example_input, example_target)
  print(f"Step: {step//1000}k")
  train_step(input_image, target, step)
  # Training step
  if (step+1) % 10 == 0:
  print('.', end='', flush=True)
  # Save (checkpoint) the model every 5k steps
  if (step + 1) % 5000 == 0:
  checkpoint.save(file_prefix=checkpoint_prefix)
```

Run the training loop:

```
fit(train_dataset, test_dataset, steps=4000)
```

The final output is as shown in the *figure 10.36*:

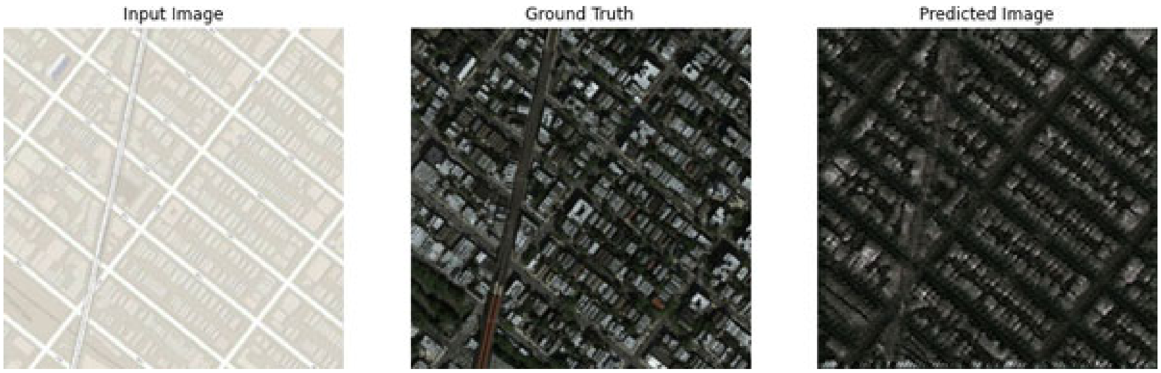


Figure 10.36: *Final predicted image after 4000 iterations*

With this, we come to the end of the section on pix2pix conditional GAN. We also created a sample that generated a real map based on.

Conclusion

In this chapter, we learnt the basics of GAN and the role of generator and discriminator. We delved into the cost function, looked at the DCGAN architecture, and how various steps like strided convolutions, batch normalization, and using ReLU as the activation function helped improve accuracy. Then, we looked at the concept of Earth movers' distance and its implementation in WGAN.

Multiple choice questions

1. **DCGAN uses Conv2D transpose.**

    a. true

    b. false

2. **Which of the following activation function is used in WGAN discriminator?**

    a. Sigmoid

    b. LeakyReLU

3. **In WGAN, the generator loss function is better than DCGAN.**

    a. true

    b. false

Answers

1. **a**
2. **b**
3. **a**

Code listing

- **https://github.com/rajdeepd/tensorflow_2.0_book_code/tree/master/ch10**
- **https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch10/dcgan_fashion_mnist.ipynb**
- **https://github.com/rajdeepd/tensorflow_2.0_book_code/blob/master/ch10/pix2pix_maps_v1.ipynb**

References

- **Build-basic-generative-adversarial-networks-gans**: **https://www.coursera.org/learn/build-basic-generative-adversarial-networks-gans/home**
- **DCGAN**: **https://arxiv.org/pdf/1511.06434.pdf**
- **WGAN-GP**: **https://arxiv.org/abs/1701.07875**
- **How to Implement Wasserstein Loss for Generative Adversarial Networks**: **https://machinelearningmastery.com/how-to-implement-wasserstein-loss-for-generative-adversarial-networks/**
- **pix2pix: https://www.tensorflow.org/tutorials/generative/pix2pix**
- **Image-to-image translation with conditional adversarial networks by Isola et al. (2017): https://arxiv.org/abs/1611.07004**

Index